AN EXPERIMENTAL IMPLEMENTATION FOR PREFIX B-TREE AND ASSOCIATED DYNAMIC LISTS

BY

HWEY-HWA WUNG

Bachelor of Education

National Taiwan Normal University

Taipei, Taiwan

Republic of China

1978

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the Degree of
MASTER OF SCIENCE
July, 1983

AN EXPERIMENTAL IMPLEMENTATION FOR

PREFIX B-TREE AND ASSOCIATED

DYNAMIC LISTS

Thesis Approved:

James R. VanDoren

Thesis Adviser

Michael J. Folk

John P Chandler

Norman N. Durham

Dean of Graduate College

PREFACE

This thesis deals with an experimental implementation of a combined B-tree indexing scheme and a buddy system for organizing and managing key words and inverted lists respectively. Measurements are developed to compare the performance of buddy system variations in terms of execution time and storage utilization.

I wish to express my sincere appreciation and thanks to Dr. James R. Van Doren, my thesis advisor, for his guidance, patience, encouragement, and understanding in the preparation of this study. Gratitude is expressed to Dr. John Chandler and Dr. Michael Folk for serving as members of the advisory committee.

I would like to thank Terry D. Kinzie and all programmers in the department of Agricultural Economics for their help and understanding. Deep appreciation is also expressed to Miss Linda Crowly for all her help in improving the readability of this thesis.

Special appreciation is due my parents, Mr. and Mrs. Chih-Shien Wung, and their son, Pey-Min, for their constant love, support, and encouragement throughout my studies in the United States.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

As an increasing number of applications require the assistance of electronic data processing, more and more information is being stored in computers. A major goal for computer scientists is to find efficient techniques to store information, especially when the information is stored in files with large, varying-size keys on secondary storage. Many different file organization techniques have been proposed. Most of those techniques that work well for smaller or formatted files do not work well for larger or minimally formatted files. The choice of a good file organization for files with minimal formatting depends on the efficiency of secondary storage utilization and the speed of information retrieval.

This thesis concentrates in part on the implementation of Prefix B-Tree, a variant of a B-tree. It has been widely used in file management, such as IBM's VSAM, Tree-Structured file directories, and textual databases. It has been found to yield good performance. The management of storage space for inverted lists associated with keys is addressed with equal importance.

Inverted lists are the lists of all records having a

given value of some attribute. For example, in textual databases, the inverted list of each referenced word contains the list of addresses on which this word is mentioned. Since the textual databases are fairly large in practical applications, it is very important to implement an efficient method to construct the words and associated inverted lists. Firstly, the organization of the index of words should be considered so as to speed up information retrieval and save space. The simple prefix B-tree and prefix B-tree, which are discussed in Chapter II, are very suitable for this consideration. Secondly, the storage of the inverted lists should be considered so as to upgrade space utilization. The number of occurrences of a word in a document varies from one word to another. This variability indicates that the lengths of inverted lists are variable. Traditionally, the associated information is stored immediately after the keys. The use of a B-tree structure will guarantee 50 percent storage utilization. Nevertheless, an alternate treatment of the associated variable-length inverted lists can further improve the storage utilization by using the buddy system, which is discussed in Chapter III.

An experimental implementation involving both a simple prefix B-tree structure and a specific dynamic storage technique is the main topic of this thesis. The Generalized Fibonacci Buddy System is used to manage a separate storage area for the inverted lists. This implementation is based

on the Prefix B-Tree of Bayer and Unterauer (1) and the Generalized Fibonacci Buddy System of Hinds (11).

Chapter II contains a discussion of the characteristics and evalution of Prefix B-Tree structures.

Chapter III presents the dynamic storage management concept and examines the basic characteristics and the dynamic storage management algorithms of the buddy systems. The variations of the buddy systems, especially the Generalized Fibonacci Buddy System developed by Hinds (11), are discussed. Finally a brief comparsion of buddy systems is addressed.

Chapter IV presents the design and implementation of simple prefix B-trees and their associated dynamic lists. The data structure design and the high level description of the implementation for both the simple prefix B-tree structure and the dynamic storage management are included.

The final chapter is a discussion of the experimental results, advantages, and disadvantages of this implementation and the practicability in file management. Possible improvement and further study are also suggested in this chapter.

Appendices include the low and high level Program Design Language (PDL) descriptions of all programs and the test size tables used for this implementation.

CHAPTER II

PREFIX B-TREE INDICES

For a given file stored on an external rotating memory device such as a disk or drum, the time required to retrieve information from the file is the main component of the total time required to process the data. An index can speed information retrieval by directing the search to the small part of the file containing the desired item. The tree-structure index has been proven effective for use with large files (6).

In 1972 Bayer and McCreight (6) first proposed the B-tree by increasing the branching factor of a binary tree from two to m to cut down dramatically the number of tree levels. Some time later, many important refinements and variations, such as the B^+-tree, simple prefix B-tree, and prefix B-tree, were explored and have become common file organizations for the storage of information on secondary storage. This chapter, which assumes that the reader is familiar with the basic B-tree and B^+-tree (5, 9, 16, 21), centers on the simple prefix B-tree and prefix B-tree. These trees are good file organizations for files with large variable-length keys.

Motivation of Prefix B-Tree Indices

Usually, B-tree schemes are used in cases in which the keys are of a fixed length. But in many applications, such as in textual database environments, the keys are generally character strings of variable length and occur in clusters. If a B-tree structure is applied to deal with variable-length keys, some undesirable situations may be encountered. For example, if a tree structure is set up with fixed-length key fields, then space wasted in the key fields and/or ambiguous decoding may occur. On the other hand, a tree structure with variable-length key fields has the advantage of avoiding both wasted space in the key fields and the ambiguous decoding of the keys, but has the disadvantage of storing the prefixes repeatedly. Additionally, in a B^+-tree, the B^+-index serves merely as a guide to direct the search to the correct leaf. It is not necessary to contain complete keys in the index nodes if the key can be represented partially, yet sufficiently enough to uniquely locate it in the leaf nodes. Therefore, the key compression techniques, front and rear compression (1, 6, 20), can be used to eliminate those characters that are not necessary to distinguish a key from the keys immediately adjacent to it. This fact implies that the use of the resultant compressed value, namely the prefix or separator, to build up the B^+-index tends to increase the degree of branching, decrease the height of the index, and save space required by the index. In 1977, Bayer and Unterauer (1) considered key

compression and proposed a refined structure, a Prefix B-Tree, to store the prefixes in the upper index part of a B^+-tree. Two kinds of prefix B-trees, simple prefix B-trees and prefix B-trees, both described by Bayer and Unterauer (1), are discussed in detail in the subsequent sections.

Simple Prefix B-Trees

Consider the rear compression technique. Bayer and Unterauer (1) suggest the technique of choosing the shortest separator, instead of using the complete key, to separate two adjacent leaf nodes so as to efficiently utilize storage space when dealing with keys of variable length. Suppose that a leaf in a B^+-tree is full and contains the keys 'compression', 'key', 'result', 'separator', and 'short', as shown in Figure 1. In order to insert the key 'rear', this leaf node must be split into two. Instead of storing the duplicated key 'result' into the upper index as usual, any string s with the property

$$\text{rear} < s \leq \text{result}$$

can be selected for the same purpose. According to the prefix property defined by Bayer and Unterauer (1), the one selected in the simple prefix B-tree approach is the prefix of the larger key of a key pair. Its length should be as short as possile. As mentioned by Bayer and Unterauer (1), this technique is only allowed when the leaf node is being split. For the index node, the same splitting technique as used in the original B-tree is applied; that is, one of the

separators on that index node is moved up one level and no further compression will be performed. Thus far, a simple prefix B-tree can be defined as a B^+-tree in which the B^+-index is substituted by a B-tree of separators.

The purpose of choosing the shortest separators is to decrease the length of the separators and increase the degree of branching. This idea can be taken a step further by scanning a small interval around the middle of a splitting leaf to obtain a good key pair so that a mininum length of the shortest separator can be obtained. Based upon the example shown in Figure 1, allowing a split point to be chosen one key to the left or to the right of the previous spilt point yields the shortest separator 'r' or 's'. Figure 2 shows the split point chosen between 'result' and 'separator' yields 's' as the separator. This method can be applied to the leaf nodes as well as to the upper level nodes.

The operations performed on a simple prefix B-tree, such as searching, inserting, and deleting, are similiar to those performed on a B^+-tree with variable-length keys, except that a shortest seporator will be selected when a node is split.

An example of a simple prefix B-tree is shown in Figure 3. The keys extracted from the test data which was used for implementing the simple prefix B-tree in this thesis are inserted in the tree in the following random order: suppress, support, suspicion, suspect, tamper, term, trialy,

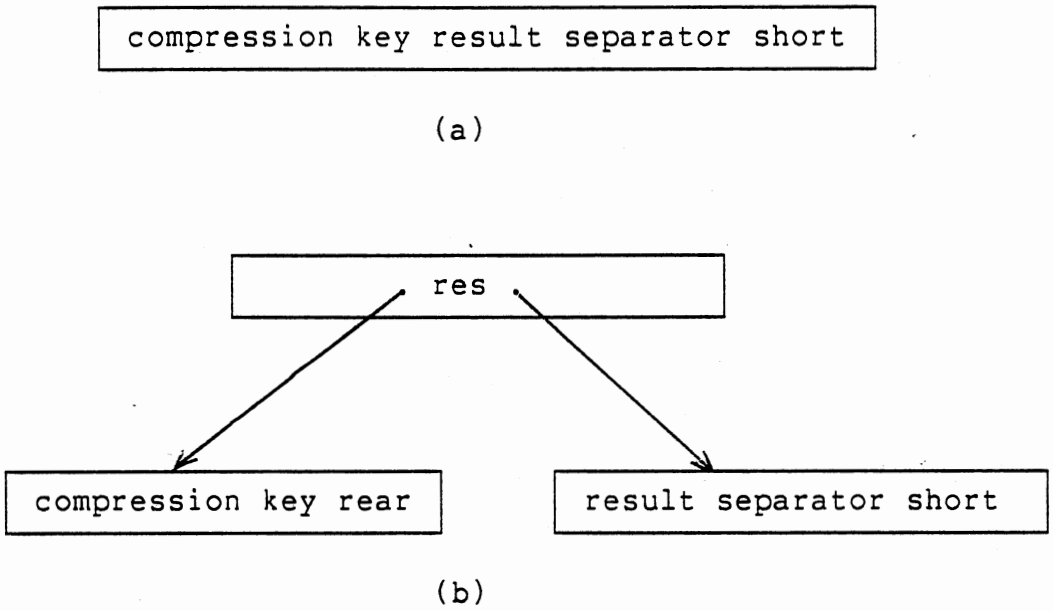


Figure 1. (a) A Full Leaf Node of B$^+$-Tree and (b) A Insertion of Key 'rear', Causing the Shortest Separator 'res' to be Promoted to the Upper Level Node

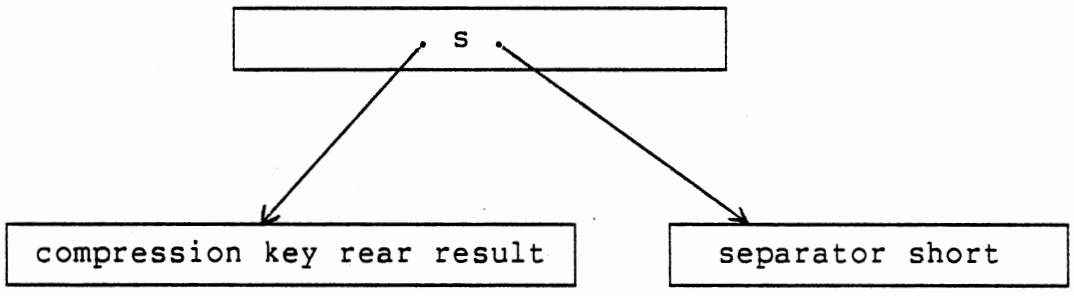


Figure 2. A Mininum Shortest Separator Chosen One Key to the Right from the Split Point of Figure 3b

supp, taylor, time, sustain, supra, surrounding, supreme, surely, sued, suppressing.

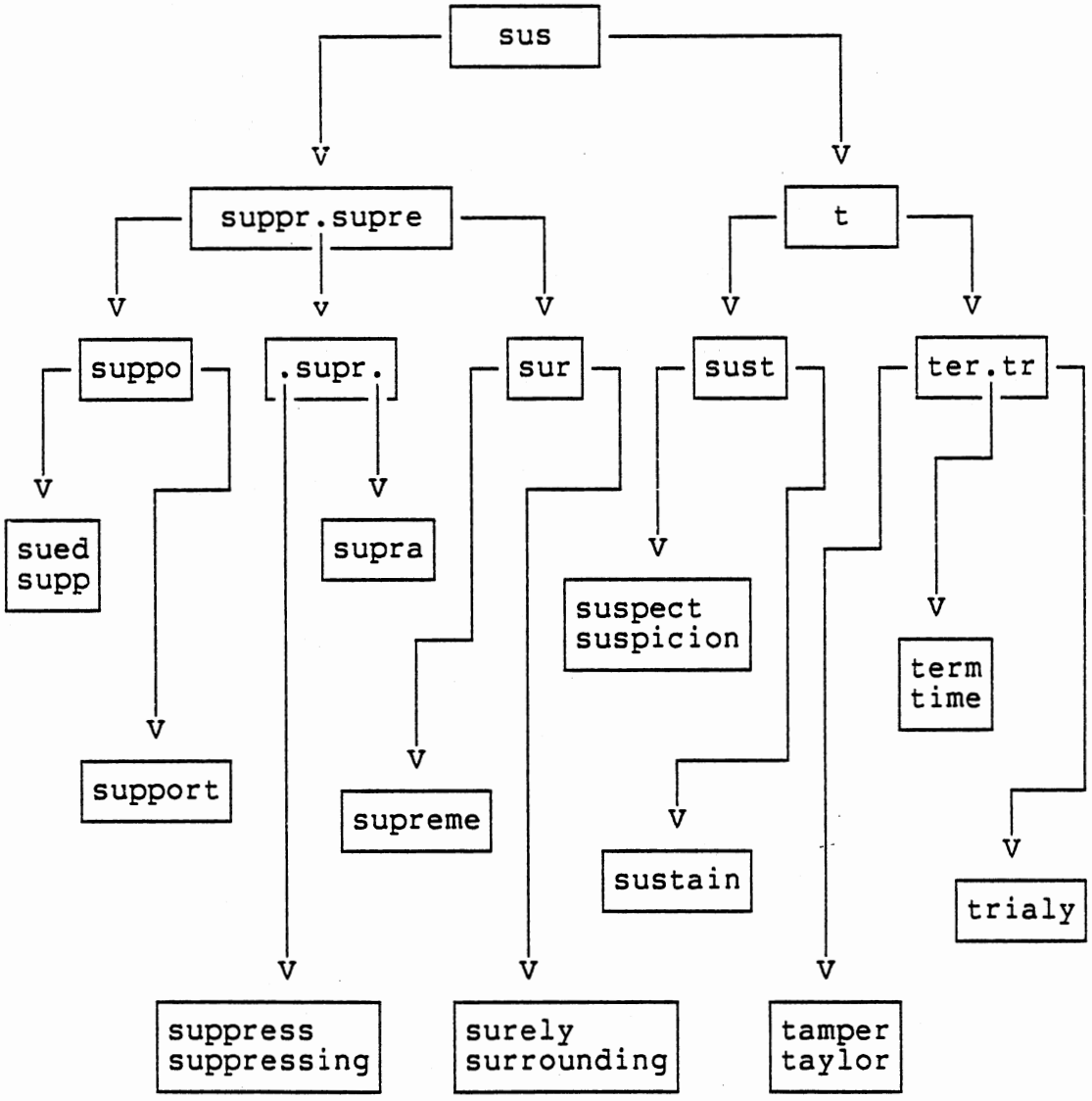


Figure 3. A Simple Prefix B-Tree Example

Prefix B-Trees

Consider again the simple prefix B-tree in Figure 3. The separators in the upper level nodes are shorter than the full keys, but the adjacent separator pairs share the common prefix which is repeatedly stored in the subtrees. Obviously, when sets of keys are in clusters, some space in the index nodes is wasted because of the repetition of the common prefixes. For better storage utilization and further reduction of the height of the index part of a simple prefix B-tree, Bayer and Unterauer (1) proposed the prefix B-tree which is based on the idea of storing the common prefix in the ancestor nodes rather than the subtrees.

For the index part of a simple prefix B-tree, suppose that node P is an arbitrary index node and that T(P) is the subtree of the index and leaf nodes with root P. The tree structure can determine the largest lower bound (LL(p)) and the smallest upper bound (SU(P)) for node P from the father node of P. For the root node of a simple prefix B-tree, assume the following:

LL(root) = the character string smaller than (chronologically preceding) any key in the tree.

SU(root) = the character string larger than (chronologically following) any key in the tree.

For all keys or separators stored in node P and/or subtree T(P), the following holds:

$$LL(P) \leq k < SU(P)$$

$$LL(P) \leq s < SU(P)$$

As shown in Figure 4, in node P, p(0),p(1), ..., p(n)

are pointers to the sons (index or leaf nodes) of node P and are denoted as son P(i) for $0 \leq i \leq n$. s(1), ..., s(n) are separators, s(n) being the last on node P. The largest lower bound and the smallest upper bound LL(P(i)) and SU(P(i)), respectively, of son P(i) for $0 \leq i \leq n$ can be defined by Bayer and Unterauer's (1, P.17) definition.

$$LL(P(i)) = \begin{cases} s(i) & \text{for } i = 1,2,\ldots,n, \\ LL(p) & \text{for } i = 0, \end{cases}$$

$$SU(P(i)) = \begin{cases} s(i) & \text{for } i = 0,1,\ldots,n-1, \\ SU(p) & \text{for } i = n. \end{cases}$$

Obviously, for any keys and separators in son P(i) there must be a common prefix c(P(i)) which can be derived by following two steps:

1. Obtain the longest common prefix $\bar{c}(P(i))$ (possibly the empty string) of the bound pair LL(P(i)) and SU(P(i)) by the front compression technique.

2. Determine the final common prefix by Bayer and Unterauter's (1, P.17) definition.

$$c(P(i)) = \begin{cases} \bar{c}(P(i))1(j) & \text{if } LL(P(i)) = \bar{c}(P(i))1(j)z \\ & SU(P(i)) = \bar{c}(P(i))1(J+1), \text{ where} \\ & 1(j) \text{ preceed } 1(j+1) \text{ immediately} \\ & \text{in the collating sequence and z} \\ & \text{is an arbitary string.} \\ \bar{c}(P(i)) & \text{otherwise.} \end{cases}$$

The largest lower bound and/or the smallest upper bound may be changed while performing insertions or deletions. As a result, the partial separators must be recomputed. Thus, the basic operations performed on a prefix B-tree are much more complicated than those of the other B-tree schemes.

As an example, again consider the tree in Figure 3. The common prefixes, shown in Figure 5, can be derived and

pruned off to yield a prefix B-tree which is illustrated in Figure 6.

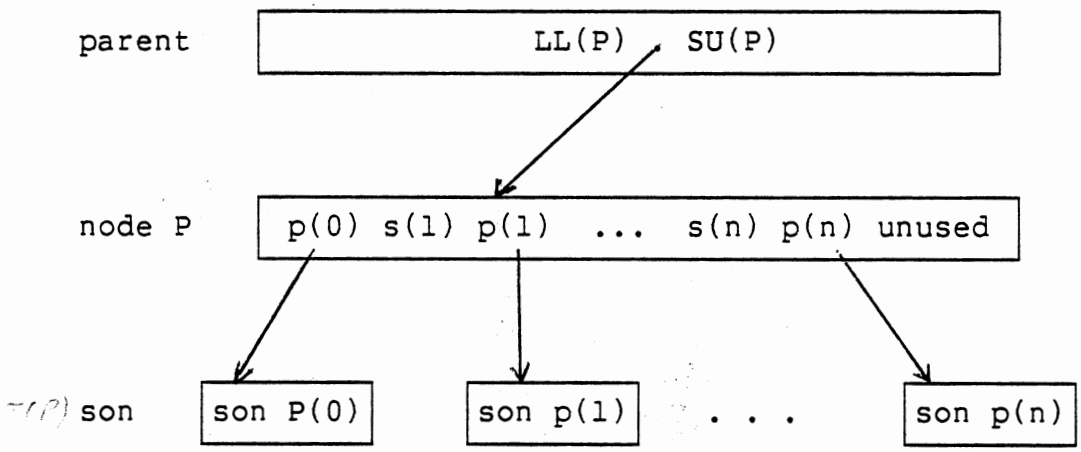


Figure 4. Partial Index Structure of a Simple Prefix B-Tree

bound pair	front compression	common prefix
suppr,supre	sup	sup
supre,sus	su	su
sus,t	empty string	s

Figure 5. Common Prefix in Simple Prefix B-Tree of Figure 5

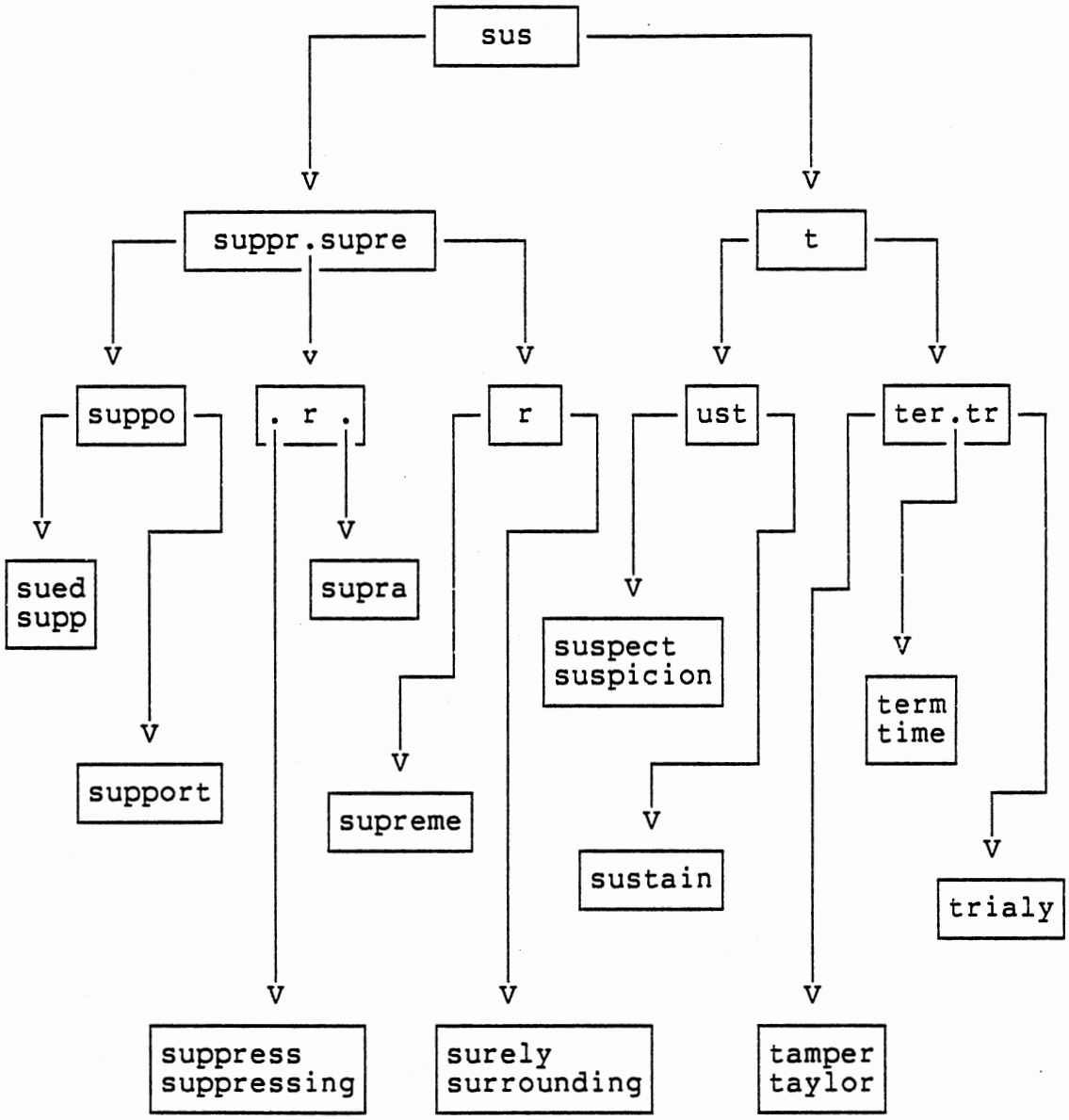


Figure 6. Prefix B-Tree Derived from the Simple Prefix B-Tree of Figure 3

Evaluation of Prefix B-Tree Indices

Since both simple prefix B-tree and prefix B-tree are variations of a B^+-tree, the main advantages of the B^+-tree, such as guaranteeing good worst-case performance, good storage utilization, and easy sequential processing, are preserved. The techniques of choosing the shortest separators, pruning off prefixes, and constructing prefixes during a search are applied on both trees. Therefore, the number of index levels, the number of disc accesses, the retrieval time, and the space required by both trees are less than those of B^+-tree. However, the index building and maintenance processes are quite complicated and time consuming, especially for the prefix B-tree.

The separators in the index nodes are of variable length, so that the branching degree of each node depends heavily on the internal organization of a node. Thus, during the index building and maintenance process, the number of separators that can be packed into a node will not be known until the predetermining tests have been performed. Additional internal searching time is required due to the varying location of separators within a node. For a prefix B-tree, additional computation time is required for both (1) recalculating partial separators for some insertions or deletions which may alter the common prefix and (2) constructing prefixes while traversing the prefix B-tree during a search.

CHAPTER III

GENERALIZED FIBONACCI BUDDY SYSTEM

In pratical applications, various amounts of memory space are required for accommodating many requested sets of information (data arrays, programs, etc.) concurrently in main memory or for storing information on secondary storage. However, the capacity of the computer is limited. Poor storage management may cause the available memory to be scattered throughout the memory pool so that the computer can not allocate space for larger contiguous memory requests. Therefore, managing or utilizing computer storage, both internally and externally, in an efficient manner is one main aspect of modern computing.

Basically, there are two types of management methods: static storage management and dynamic storage management. Static storage management allocates the storage blocks in fixed sizes while dynamic storage management allocates the storage blocks in varying sizes. This chapter discusses dynamic storage management, which is the better method for storing variable-length lists. It focuses on one particular category of this management -- the buddy-system method.

Dynamic Storage Allocation

Many applications need blocks of varying sizes that share a common memory area. Dynamic storage allocation techniques are required for dynamically allocating (reserving) and deallocating (releasing or freeing) variable-size blocks of contiguous memory cells from a common storage pool.

Obviously, storage blocks are divided into two classes: free and reserved. When an area of n consecutive free space is requested, a block of the appropriate size is selected from the common storage pool and becomes a reserved block. When a reserved block is released, this block is returned to the common storage pool and becomes a free block. These processes are the fundamental concepts of allocation and deallocation. Based upon these ideas, several methods for dynamic storage allocation have been published. The various methods follow different procedures for gaining an available block and returning the excess storage of this block. Some common methods are given in this section. It is intended that this section explains why the buddy system is chosen to manage a separate memory space for information associated with words in the Prefix B-Tree experimental implementation.

First-Fit

In the first-fit policy, the free blocks are linked into a circular list in some order, such as in ascending or descending order of block addresses, in order of block size,

or in random order. When a request for a block of size n words is serviced, a search is made along the free ring until the first block of size m ≥ n encountered. This block is then detached from the list. The starting search point can be the beginning of a free list, or it can circulate to the right around the ring. If m >> n, the block is split into two small blocks, one of size n, which is marked as reserved and satisfies the request, and one of size m-n, which is marked as free and is put back on the available list. When a block is liberated, an attempt to coalesce this block with its neighbors is made to form a larger free block. The resulting, and possibly enlarged, free block is put back on the free list. Sufficient information, such as block size and block class (free or reserved), must be carried in each block for the operation of coalescence (15).

Best-Fit

The best-fit method, like the previous method, employs a circular list of all available blocks. When a block is requested, a search of the entire list is performed to find the smallest block that is large enough to fulfill the request. The excess part of the block, if any, is put back on the free list. When a block is liberated, the same coalescing technique as used in the first-fit method is applied.

Buddy-System

This scheme breaks memory into blocks of prescribed sizes, such as blocks whose sizes are powers of two or blocks whose sizes are numbers in the Fibonacci sequence. Blocks with sizes in these sequence schemes can be split into two smaller blocks, namely buddies, whose sizes are also numbers in the sequence schemes and also can be reconstituted if and when both buddies are simultaneously free. Additionally, the free blocks of the various sizes are placed on the doubly linked lists of blocks of the same size. Therefore, when a block is allocated, only the available list containing blocks of sizes equal to the requested size is examined. If this list is not found, then list with the next larger block size is examined. As stated in Hinds' (11, p.221) article, the following actions are performed when the operation of allocation or deallocation is encountered.

A: To satisfy a storage request

1. The smallest block of storage that is at least as big as the request is selected as the candidate block.

2. The candidate is checked for size and, if large enough, is split into two smaller blocks (buddies); otherwise the candidate block is returned as the block satisfying the request and the algorithm terminates.

3. One of the buddies (the smaller of the two, if possible) is selected as the new candidate and the other is inserted into the free storage pool. The algorithm then proceeds from A.2.

B: To return a block to the storage pool

1. The buddy of the newly returned block is located.

2. The buddy is inspected to see that it is whole (not split into subbuddies) and free. if both conditions are met, the the buddy is removed from free storage and merged with the newly returned block to create a larger block. This larger block is then taken as the newly returned block and execution proceeds with B.1.

3. If it is impossible to merge, then the newly returned block is returned to the free storage area and the algorithm terminates.

Since the buddy system manages blocks of storage on separate availability lists rather than managing a single availability list as the other methods do, the number of searches per request of the buddy system is less than those of other dynamic storage methods. Thus, use of the buddy system on secondary storage is motivated by the speed of finding a block and by the speed of allocating and deallocating a block.

Characteristics of Buddy System

The buddy system was first published by Knowlton (14) in 1965. It was used for the storage bookkeeping method in the Bell Telephone Laboratories Low Level List Language. Since that time, there have been evolutionary systems developed from Knowlton's original buddy system. Such systems differ to some extent but all are similiar in many features. Such features are as follows:

1. Memory is broken into many sizes of blocks that are fixed in size and location.

2. Each block size has its own separate availability list.

3. The basic structures for the allocation and deallocation algorithms are the same.

The major differences are the sizes of the memory blocks provided and the consequent address calculation for locating the buddy of a released block.

As stated in the preceeding section, the buddy system has a time performance advantage over other dynamic methods. This advantage, however, is achieved at the expense of low level storage utilization due to internal fragmentation and external fragmentation. The other dynamic methods are subject to external fragmentation alone. Internal fragmentation refers to unused storage that dwells inside the reserved blocks, whereas external fragmentation refers to free blocks that are unable to service requests because they are of insufficient size.

Original Buddy System

In Knowlton's original buddy system, namely the binary buddy system, the lengths of the blocks are of powers of two and contain two control fields and/or two link fields, forward links and backward links. One control field, TAG, is used to indicate if the block is free or in use. If the block is free the two links are provided to link free blocks into a ring. Otherwise, the space for the links is used for storing information. The other control field, ISIZE, used to contain the base 2 logarithm of the block size. When a block is requested or liberated, the algoritms presented in the preceeding section are used.

The buddy location process is relatively simple. Suppose the entire pool of memory space consists of 2^m words, which are assumed to have relative addresses 0 through $2^m - 1$. The block address for a block of size 2^k is a binary number in which the last k bits are zero. For example, a block of size 16 has an address of the form bb...b0000 (where the b's represent either 0 or 1). If it is split, the newly formed buddies of size 8 have the addresses bb...b0000 and bb...b1000. Hence, given the address bb...b000 of a block of size 8, the address of its binary buddy can be obtained by complementing the fourth bit from the last bit. In general, given the address of a block of size 2^k, the address of its buddy is obtained by complementing the (k+1)st bit from the last.

Since storage is allocated in blocks of fixed, uniform size by the buddy system, a request for memory is forced to be rounded up to nearest block size. Therefore, unusable memory occurs in fragments both internal and external to the allocated blocks. Internal fragmentation, however, poses a larger problem than external fragmentation (19). Generally speaking, the more different-size blocks there are available, the less internal fragmentation there should be. Hence, this leads to investigations into methods that allow more block sizes, such as the Fibonacci system by Hirschberg (12), the weighted buddy system by Shen and Peterson (22), and the generalized Fibonacci buddy system by Hinds (11).

Fibonacci System

The sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...,

in which each number is the sum of the preceeding two, was originated in 1202 by Leonardo Fibonacci. The numbers in the sequence are denoted by F(n), and are formally defined as

$$F(0) = 0, \quad F(1) = 1,$$

$$F(n+2) = F(n+1) + F(n) \quad n \geq 0.$$

This sequence was given the name "Fibonacci Numbers" by a mathematician named E. Lucas during the 19th century. The name has been used ever since (15).

Based upon the Fibonacci number, a new system, namely the Fibonacci system, was introduced by Hirschberg (12) in 1973. This system possesses the basic characteristics of the buddy system. It has, however, its own set of permissible block sizes, which are based on the numbers in the Fibonacci sequence, and its own buddy locating process. Since the calculation of the possible starting addresses of the buddies needs three extra auxiliary lists and is a time-consuming computation, the buddy locating process is not addressed in this thesis.

Weighted Buddy System

The weighted buddy system, introduced by Shen and Peterson (22) in 1974, permits block sizes of 2^k, $0 \leq k \leq m$, and $3 \cdot 2^k$, $0 \leq k \leq m-2$. In this system there are nearly

twice as many block sizes as there are available in a binary buddy system. As suggested by Shen and Peterson (22), the blocks are split in two different ways depending on the size of the block to be split: (1) a block of size 2^{k+2} is split into two blocks of sizes $3 \cdot 2^k$ and 2^k, or (2) a block of size $3 \cdot 2^k$ is split into two blocks of sizes 2^{k+1} and 2^k.

To distinguish these different kinds of splits, a two-bit TYPE field is encoded in each block:

TYPE(P) = 11 if the block with address P is split from a 2^k size block,

= 01 if the block with address P is the left split from a $3 \cdot 2^k$ block,

= 10 if the block with address P is the right split from a $3 \cdot 2^k$ block.

Given the size k and the address x of the block, the address calculation for the buddy of this block is defined as: (22, p.560)

buddy (x) = $x + 3 \cdot 2^k$ if $x \bmod 2^{k+2} = 0$ and TYPE(x)=11,

= $x - 3 \cdot 2^k$ if $x \bmod 2^{k+2} = 3 \cdot 2^k$ and TYPE(x)=11,

= $x + 2^k$ if TYPE(x)=01,

= $x - 2^{k+1}$ if TYPE(x)=10.

Generalized Fibonacci Buddy System

In 1975, a generalized Fibonacci buddy system was published by Hinds (11). There are two innovations in this system. The first is that block sizes are allowed to be elements of an arbitrary (but fixed) generalized Fibonacci sequence. The sequence has the form

$$F(n) = F(n-1) + F(n-k)$$

Thus, any sequence for a buddy system can be completely generalized by choosing k and F(i) ($0 \leq i \leq k-1$). For instance, k=1 and F(0)=1 yield the original buddy block sizes:

1, 2, 4, 8, 16, 32, ...

K=2, F(0)=3, and F(1)=5 yield the Hirschberg's Fibonacci buddy block sizes:

3, 5, 8, 13, 21, 34, ...

The second innovation is a very simple and efficient technique for locating buddies by using the concept of Left-Buddy-Count (LBC).

The LBC can be described as follows. Suppose that an arbitrary Fibonacci sequence has been picked and is to be used on a storage pool. After a period of time in which allocations and liberations have taken place, the pool may be broken up and the resulting storage block with the LBC field would appear as shown in Figure 7. As stated by Hinds (10), each vertical line indicates the start of a block, each horizontal line indicates a split and also points to the right-hand buddy. Additionally, this diagram indicates exactly what merges would have to take place and what blocks are to be inspected to achieve the merges. Thus, to make the merging process easy, the number of splits a block has undergone, since it was created, must be encoded into the blocks at the time of a split. This number is called the Left-Buddy-Count (LBC).

The LBC is utilized with the other attributes of a

block, such as FREE, ISIZE, and K. The FREE field is a boolean value indicating the availability of a block. The ISIZE field is an index into a vector SIZE containing the actual size of the block. K is a constant used to locate buddies and assign proper sizes during a split. When a block of size F(n) is split, the block of size F(n-1) is the left buddy and the block of size F(n-k) is the right buddy.

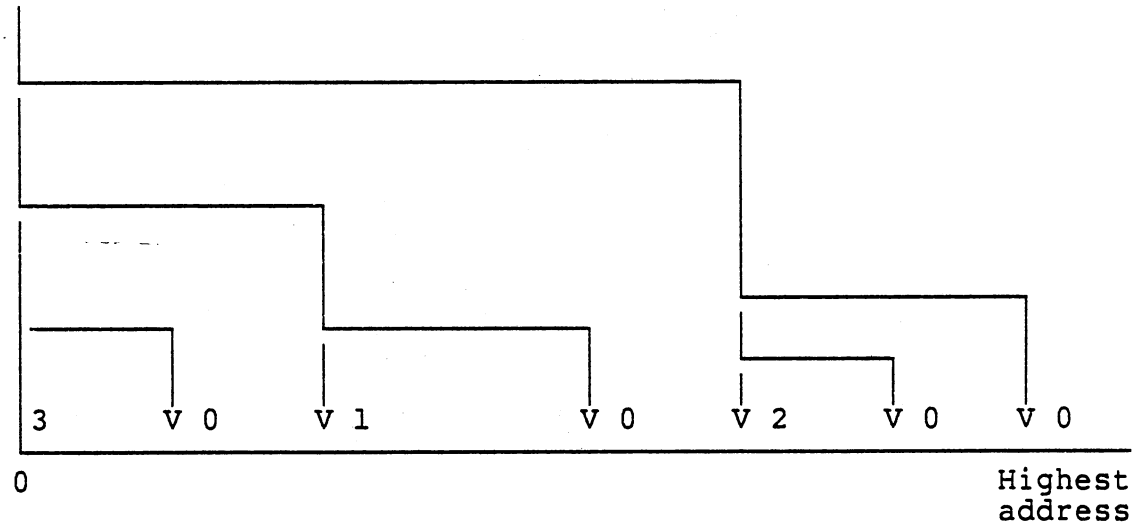


Figure 7. Hinds' Fibonacci System Storage Layout with Left Buddy Count

According to Hinds' (11, P.222) definition, to assign the proper LBC to a block, let the LBC of the entire storage pool be 0 at the beginning. During any split of a parent block, assign to the newly created blocks:

LBC(right) = 0

LBC(left) = LBC(parent) - 1

For a merge the reverse relation is used:

LBC(parent)= LBC(left) - 1

Therefore, the determination of the relative location of a buddy is easily done by testing the LBC. If LBC=0 it is a right buddy; otherwise, it is a left buddy. To locate buddies, the following formules are used:

For the left-handed buddy

ISIZE = liberated block's ISIZE + K - 1
address = liberated block's address -
SIZE(left-handed buddy's ISIZE)

For the right-handed buddy

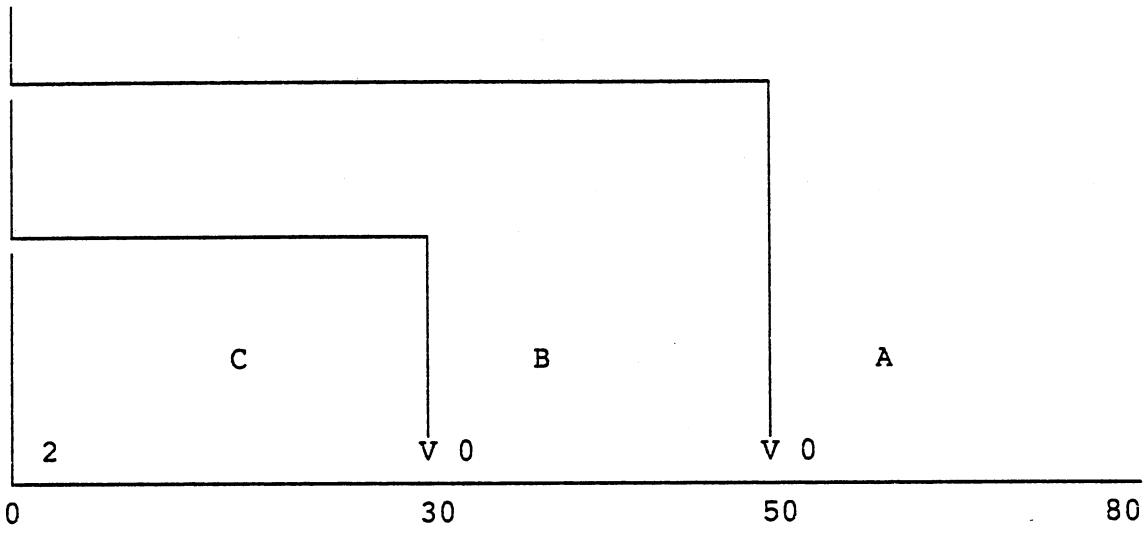
ISIZE = liberated block's ISIZE - K + 1
address = liberated block's address +
SIZE(liberated block's ISIZE)

For example, as shown in Figure 8, if block A whose address, ISIZE, and LBC are 50, 3, and 0, respectively, is liberated the following steps are performed:

1. Test LBC(A) to determine if its buddy is a left-handed buddy.
2. Compute ISIZE for the left-handed buddy
   ISIZE = 3 + k - 1 = 4
3. Compute address for the left-handed buddy
   address = 50 - SIZE(4) = 0

On the other hand, if block C whose address, ISIZE, and LBC are 0, 3, and 2, respectively, is liberated, the address and ISIZE of its buddy are computed as follows:

1. Since LBC(C) > 0, its buddy is right-handed.
2. ISIZE of C's buddy = 3 - k + 1 = 2
3. address of C's buddy = 0 + SIZE(3) = 30



SIZE = 10, 20, 30, 50, 80
k = 2

Figure 8. An Example of Hinds' Buddy System Buddy Locating Process.

When merging two buddies together, two conditions should be confirmed: both buddies are simultaneously free, and both are not subdivided. As an example consider the storage layout in Figure 8. Block A can not be merged with its buddy because its buddy is already divided into block B and block C.

Buddy System's Variation for Disk Allocation

Secondary storage has traditionally been managed with a static storage technique. The buddy system has been proven effective in internal storage (15). Therefore, Burton (3) was interested in the buddy system for possible application on secondary storage. However, in Hinds' buddy system the block sizes are the form $F(n) = F(n-1) + F(n-k)$ where k is a simple constant. This method tends to be unsuitable for disc storage allocation since logical blocks often overlap the boundaries of physical blocks, such as sectors, tracks, cylinders, and disc packs. Hence, in 1975, Burton (3) improved this method by selecting a meaningful integral-value function k, so that the set of permitted block sizes are the form $F(n) = F(n-1) + F(n-k(n))$.

The merits of Burton's (3, P.416) improvement are as follows:

1. It is possible to prevent any logical block from overlapping the boundary of any physical block which is larger than the logical block.

2. Any sequence of block sizes may be allowed.

These merits can be illustrated by Burton's (3, P.417) examples. Suppose that some ICL discs have 128 words per sector, 1024 words per track, and 10,240 words per cylinder. Table I shows one sequence of block sizes which insures that every physical block is also a logical block. Table II illustrates how, if need be, blocks of size 50 and 150 could be provided while using the ICL discs.

The splitting and coalescing processes can be performed

in the usual manner. However, the k in this new variation of the buddy system is a function rather than a constant. Thus, a new problem of determining the size of the buddy of a right block occurs during coalescing. To solve this problem, Burton (3) introduces a new field, called the I-field, to the control word of each block. The information contained in the I-field concerns the sizes of the buddies of the right blocks. For a right block, the I-field contains the index of the size of the block's buddy. For a left block, the I-field contains the value of the I-field of its parent block. Consequently, the size of a right block's buddy can be preserved when the block is split.

TABLE I

A FIBONACCI-LIKE SCHEME (k(i) OFTEN 2)

i	SIZE(i)	k(i)
0	16	-
1	32	1
2	48	2
3	80	2
4	128	2
5	256	1
6	384	2
7	640	2
8	1024	2
9	2048	1
10	4096	1
11	6144	2
12	10240	2

TABLE II

AN IRREGULAR SCHEME

i	SIZE(i)	k(i)
0	22	-
1	28	-
2	50	2
3	78	2
4	106	3
5	128	5
6	150	6
7	256	3
8	384	3
9	640	2
10	1024	2
11	2048	1
12	4096	1
13	6144	2
14	10240	2

Comparison of Buddy Systems

There are two properties, running time and storage utilization effectiveness, which are important for a buddy system. The running time is determined by the number of blocks which are split and recombined. The efficiency of storage management is analyzed in terms of internal and external fragmentation. When a requested block size is not equal to one of the provided block sizes, it is necessary to

allocate the next larger block size for this request. The sum of unusable memory due to this overallocation over all allocated blocks is referred to as internal fragmentation. Memory overflow occurs when the requests can not be satisfied because the available blocks are of insufficient sizes. The ratio of the amount of unallocatable memory to the total memory size is referred to as external fragmentation. Since the external fragmentation is a proportion of total memory and the internal fragmentation is a proportion of allocated memory, total fragmentation should be computed as follows:

total = (1-external) * internal + external

According to these measures, a simulation of four buddy systems (binary, Fibonacci, weighted, and the F-2 buddy system based on the recurrence relation F(n+1)=F(n)+F(n-2)) was conducted by Peterson (18) in 1977 to obtain comparative values of internal, external, and total fragmentation as well as the average numbers of splits and recombinations. Note that two kinds of distributions, uniform and exponential, were used to generate the sequence of requests in this simulation. The comparative simulation results show that as internal fragmentation decreases, external fragmentation increases. This occurs because an increased number of different block sizes, such as those in the weighted and F-2 buddy systems, will result in a smaller intrablock difference (F(n) - F(n-1)). Therefore, a better fit to the requested block size can be made to yield lower

internal fragmentation. This also tends to increase the number of smaller available blocks which are less useful than the larger blocks provided by the binary and Fibonacci buddy systems. These small and unusable but available blocks contribute to higher external fragmentation. However, the total fragmentation for those buddy systems is relatively constant, resulting in 25 to 40 percent of the memory being wasted. The running time increases with an increase in external fragmentation.

The amount of internal and external fragmentation in a buddy system depends heavily on the distribution of requests for memory and the block sizes provided. It is impossible to change the memory distribution to match the allocation method. But, it would be possible for a buddy system to generate a sequence of numbers more closely matched to the storage allocation requirement.

CHAPTER IV

PREFIX B-TREE INDEX AND ASSOCIATED DYNAMIC LISTS IMPLEMENTATION

An experimental implementation for a simple prefix B-tree and a dynamic storage method is presented in this chapter. A simple prefix B-tree structure is set up for storing the variable-length words. One extension of Hinds' and Burton's generalized Fibonacci buddy system is the system used to deal with inverted lists which are associated with words. Program Design Language (PDL) descriptions of this implementation are available in the appendices.

The implementation language for this project is PL/I. In PL/I, a REGIONAL organization of a data set divides the data set into regions and permits the users to control the physical placement of regions in the data set. Because of these features, relative address files are used in this implementation. Each physical region will be treated as either one leaf node or index node in the tree structure or one common storage pool in the buddy system. The efficiency of storage utilization, data transmission time, and internal searching time depend on the region size. This thesis will not study the topic of region size any futher and fixes the size to 1024 bytes.

Implementation for Simple Prefix B-Tree

Data Structure Design

The PL/I relative address file used for the tree structure portion is called TREE. Since both the actual keys in the index nodes and the separators in the leaf nodes are variable in length, some additional information must be kept in each node so that the fields and key values can be located correctly. The data structures of the leaf nodes and the index nodes are shown in Figure 9. Figure 10 presents the declarations of major data structures.

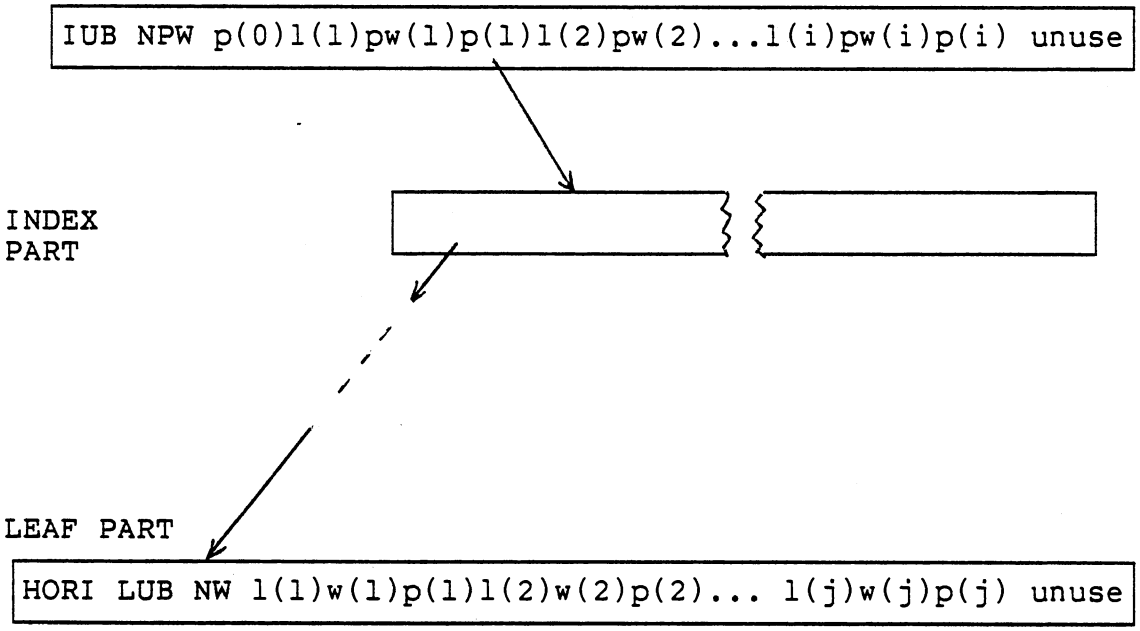


Figure 9. Data Structures for a Simple Prefix B-Tree

The index node and leaf node are called I_REG and L_REG, respectively. Both I_REG and L_REG are vectors of bytes that hold regions of information stored in the TREE. The additional information kept in I_REG includes IUB, NPW, and l(i), which represent the number of bytes unused, the number of separators stored within this index node, and the length of separators pw(i), respectively. Likewise, LUB, NW, and l(j) in the leaf node represent the number of bytes unused, the number of actual words stored within the leaf node, and the length of w(j), respectively. In the index nodes, p(i) is a pointer to a descendant node as usual. However, in the leaf node, p(j) is a pointer to the inverted list associated with w(j). Since each physical region in the relative address file is identified by a region number, IREG_NUM or LREG_NUM is used to represent the region number, depending on whether this region is used as an index node or leaf node. IREG_DISP and LREG_DISP are the offset within I_REG and L_REG, respectively. Except when a node is being searched and updated, I_INFO, L_INFO, RPOINT, and APOINT are dynamically based on I_REG(IREG_DISP), L_REG(LREG_DISP), I_REG(IREG_DISP+2+PW_LEN), and L_REG(LREG_DISP+2+WORD_LEN) accordingly.

Logical Design

This implementation centers on the random insertion process of building and maintaining a simple prefix B-tree structure. Therefore, only the insertion algorithm is

designed. The overflow condition is handled by simply splitting the overflow node into two or by attempting to share the overflow node with one of its brothers. When a node shares with its brother, the right brother is used unless it is full, in which case the left brother is used.

```
TREE FILE RECORD ENVIRONMENT(REGIONAL(1),RECSIZE(1024));

I_REG(1024)          CHAR(1);
L_REG(1024)          CHAR(1);

1  I_HEAD   BASED(I_REG(1)),
     2  I_UNUSE_BYTE          FIXED BIN(15,0),
     2  NUM_PW                FIXED BIN(15,0);

1  L_HEAD   BASED(L_REG(1)),
     2  HORIZONTAL            FIXED BIN(15,0),
     2  L_UNUSE_BYTE          FIXED BIN(15,0),
     2  NUM_WORD              FIXED BIN(15,0);

1  I_INFO   BASED(I_REG(IREG_DISP)),
     2  LPOINT                FIXED BIN(15,0),
     2  PW_LEN                FIXED BIN(15,0),
     2  PART_WORD             CHAR(200);

RPOINT      BASED(I_REG(IREG_DISP+2+PW_LEN))FIXED BIN(15,0);

1  L_INFO   BASED(L_REG(LREG_DISP)),
     2  WORD_LEN              FIXED BIN(15,0),
     2  WORD                  CHAR(200);

1  APOINT   BASED(L_REG(LREG_DISP+2+WORD_LEN)),
     2  REG                   BIT(8),
     2  DISP                  BIT(8);

IREG_NUM,LREG_NUM            FIXED BIN(15,0);
IREG_DISP,LREG_DISP          FIXED BIN(15,0);
```

Figure 10. Declaration of Major data Structures for a Simple Prefix B-Tree

Refer to Figure 11 for the following discussion. Since the technique of choosing the shortest separator is used only for splitting leaves, the split and equalization procedures of a lowest level node are different from those of an upper level node. When an index node is being split, one of the separators in that node is moved up one level in the tree. Consequently, the split and equalization procedures of an index node are more complicated than those of a leaf node.

Since the separators are of variable length, overflows may trigger further splits, merges, or overflows if a separator in the parent node is replaced by a longer or shorter separator. However, such a condition occurs infrequently. Thus, for simplification, such a condition is treated as a equalization failure if it is detected.

Implementation for Dynamic Lists

Data Structure Design

The separate space for inverted lists is a relative address file named ADRS. The physical region in ADRS is called ADDR_REG, which is a vector of words that hold the inverted lists stored in ADRS. For simplification, this implementation uses four bytes for each address. Since the inverted lists are variable in length, several inverted lists may share an ADDR_REG or one inverted list may cross one or more ADDR_REGs. Therefore, the starting point of each inverted list in ADDR_REG is not fixed. As shown in

```
INDX_UPDATE: PROCEDURE;
    IF OVERFLOW NODE HAS RIGHT BROTHER
         THEN CALL EQUAL_LEAF, IF SUCCEEDS THEN RETURN;
    IF OVERFLOW NODE HAS LEFT BROTHER
         THEN CALL EQUAL_LEAF, IF SUCCEEDS THEN RETURN;
    CALL SPLIT_LEAF;
    DO WHILE(CURRENT OVERFLOW NODE IS NOT ROOT NODE);
         PROPAGATE THE SHORTEST SEPARATOR INTO PARENT NODE;
         IF PARENT NODE IS NOT FULL THEN RETURN;
         SET PARENT NODE TO OVERFLOW NODE;
         IF OVERFLOW NODE HAS RIGHT BROTHER
            THEN CALL EQUAL_INDX, IF SUCCEEDS THEN RETURN;
         IF OVERFLOW NODE HAS LEFT BROTHER
            THEN CALL EQUAL_INDX, IF SUCCEEDS THEN RETURN;
         CALL SPLIT_INDX;
    END;
    CREATE A NEW ROOT NODE;
END INDX_UPDATE;

SPLIT_LEAF: PREOCEDURE;
    FIND THE SPLIT POINT WITHIN A CERTAIN INTERVAL;
    SPLIT OVERFLOW NODE INTO TWO LEAF NODES;
    COMPUTE THE SHORTEST SEPARATOR BETWEEN THESE NEW
         CREATED LEAF NODES:
END SPLIT_LEAF;

SPLIT_INDX: PROCEDURE;
    FIND THE SPLIT POINT WITHIN A CERTAIN INTERVAL;
    PUT SEPARATORS BEFORE SPLIT POINT IN ONE INDEX NODE;
    SET THE SEPARATOR ON THE SPLIT POINT TO THE SHORTEST
         SEPARATOR TO BE PROPAGATED;
    PUT THE REST SEPARATORS IN ANOTHER INDEX NODE;
END SPLIT_INDX;

EQUAL_LEAF: PROCEDURE;
    IF TOTAL LENGTH OF OVERFLOW NODE AND ITS BROTHER IS
         GREATER THAN TWO TIMES OF REGION SIZE THEN RETURN;
    COMBINE OVERFLOW NODE AND ITS BROTHER INTO TOTAL NODE;
    FIND SPLIT POINT AND SPLIT TOTAL NODE INTO TWO LEAVES;
    COMPUTE AND UPDATE THE SHORTEST SEPARATOR BETWEEN THESE
         NEW CREATED LEAF NODES;
END EQUAL_LEAF;

EQUAL_INDX: PROCEDURE;
    IF TOTAL LENGTH OF OVERFLOW NODE AND ITS BROTHER IS
         GREATER THAN TWO TIMES OF REGION SIZE THEN RETURN;
    COMBINE OVERFLOW NODE, ITS BROTHER, AND THEIR SEPARATOR
         INTO TOTAL NODE;
    FIND SPLIT POINT, SPLIT TOTAL NODE INTO TWO INDEX NODES,
         AND UPDATE THE SEPARATOR IN THE PARENT NODE;
END EQUAL_INDX;
```

Figure 11. Logic of Handling Overflow Condition for a Simple Prefix B-Tree

Figure 9, the pointer, p(j), in the leaf nodes consists of the relative region number REG and the offset within the region named DISP. Refer to Figure 12 for more details concerning inverted lists.

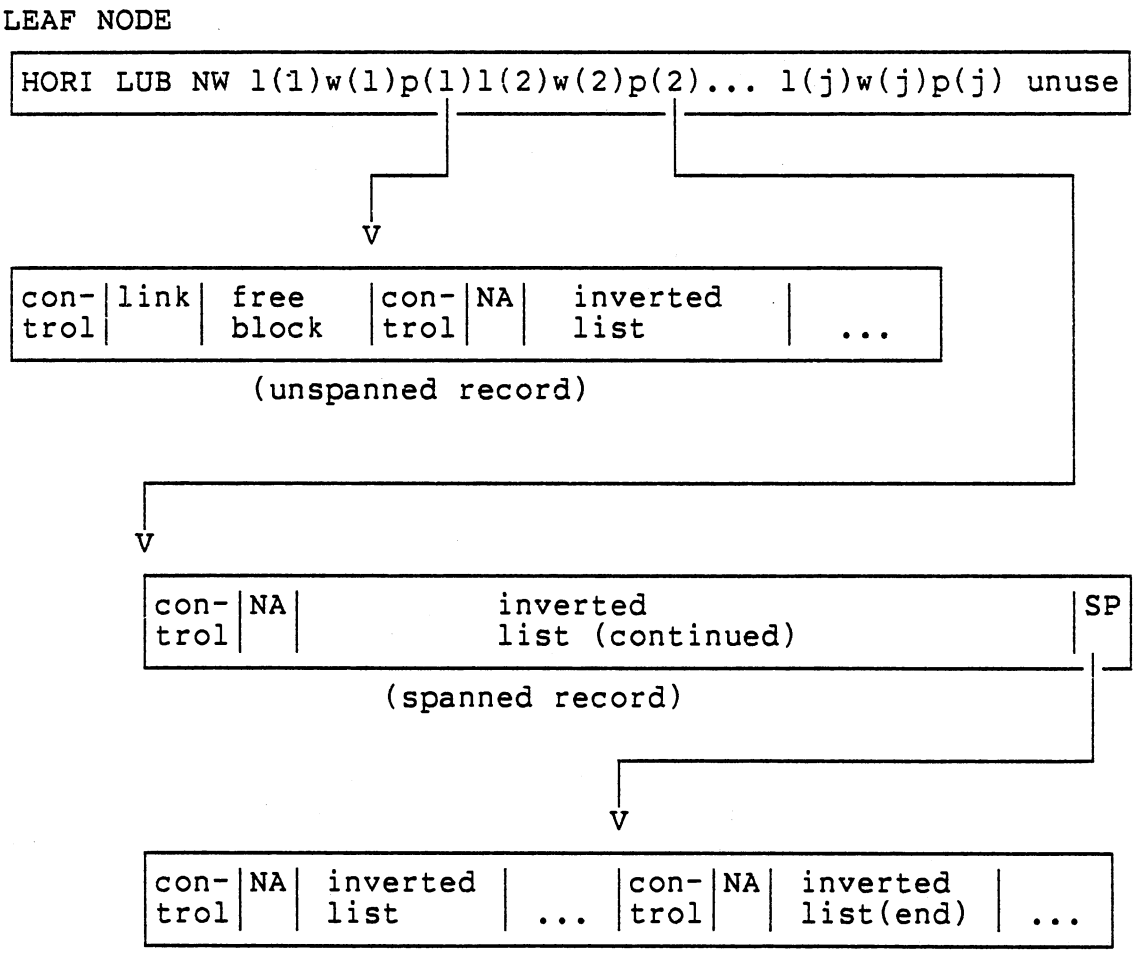


Figure 12. Data Structure for Managing Fragment Space in Using Portions of Blocks

The major data structure of the dynamic lists is HEADER

and is stored in the first region of ADRS. It is a structure that contains all pertinent run information that must be saved for future reuse. It contains SIZE_NUM, which is the number of different block sizes allowed in a common storage pool, and an available space list (SIZE_TABLE), which keeps track of all available blocks of storage. SIZE contains the various allowable block sizes in increasing order. K is the corresponding generalized Fibonacci factor. SLLINK and SRLINK are the left and right links for the headers of the availability lists. With respect to the particular application, additional information, such as ROOT, NEXT_TREE_REG, and NEXT_ADDR_REG, must be kept to indicate the region number of the root node, of the next available region for the tree structure, and of the next available region for the inverted lists. The declarations for the data structures used in the dynamic lists implementation are listed in Figure 13.

Each block needs one word for control purpose. Moreover, the free blocks need one more word for left and right links which link the blocks to the doubly linked available lists. Therefore, two based arrays, CONTROL_FIELD and LINK_FIELD, are used to redefine the control field and/or link field for a particular purpose. CONTROL_FIELD, which is based on ADDR_REG(0), has four elementary items. FREE, LBC, and ISIZE have the same purpose as in Hinds' (11) article. IFIELD corresponds to the I-field in Burton's (3) article. LINK_FIELD, based on ADDR_REG(0), contains the

```
ADRS FILE RECORD ENVIRONMENT(REGIONAL(1),RECSIZE(1024));

ADDR_REG(0:255)          FIXED BIN(31,0);
ADDR_REGB(0:255)         FIXED BIN(31,0);
ADDR_REGS(0:255)         FIXED BIN(31,0);

1  HEADER,
   2  ROOT               FIXED BIN(15,0),
   2  NEXT_TREE_REG      FIXED BIN(15,0),
   2  NEXT_ADDR_REG      FIXED BIN(15,0),
   2  SIZE_NUM           FIXED BIN(15,0),
   2  SIZE_TABLE(-1:-SIZE_NUM),
      3  SIZE            FIXED BIN(15,0),
      3  K               FIXED BIN(15,0),
      3  SLLINK,
         4  SLREG        BIT(8),
         4  SLDISP       BIT(8),
      3  SRLINK,
         4  SRREG        BIT(8),
         4  SRDISP       BIT(8);

1  CONTROL_FIELD(0:255)  BASED(ADDR_REG(0)),
   2  FREE               BIT(1),
   2  IFIELD             BIT(7),
   2  LBC                BIT(8),
   2  ISIZE              FIXED BIN(15,0);

1  LINK_FIELD(0:255)     BASED(ADDR_REG(0)),
   2  LLINK,
      3  LREG            BIT(8),
      3  LDISP           BIT(8),
   2  RLINK,
      3  RREG            BIT(8),
      3  RDISP           BIT(8);

1  LINK_FIELDB(0:255)    BASED(ADDR_REGB(0)),
   2  LLINKB,
      3  LREGB           BIT(8),
      3  LDISPB          BIT(8),
   2  RLINKB,
      3  RREGB           BIT(8),
      3  RDISPB          BIT(8);

1  SPAN_PT               BASED(ADDR_REGS(255)),
   2  DUMMY              CHAR(2),
   2  AREG_REGS          BIT(8),
   2  AREG_DISPS         BIT(8);

1  ADDR_FIELD            BASED(ADDR_REG(AREG_DISP+1)),
   2  NUM_ADDR           FIXED BIN(15,0),
   2  ADDRESS(254)       FIXED BIN(31,0);
```

Figure 13. Declarations of Data Structures for Managing Fragment Space in Using Portions of Blocks

left link and the right link, LLINK and RLINK, respectively.

The other major data structure is ADDR_INFO, which is dynamically based on ADDR_REG(AREG_DISP+1) and has two elementary items. NUM_ADDR indicates the number of addresses contained in the block. ADDRESS is a vector of words that hold the addresses in the block. The purpose of ADDR_INFO is to allow the access of a inverted list.

With the exception of inserting and deleting blocks from the avaialability lists, the link fields of the processed block and its forward or backward block which links or will be linked to the processed block need to be in internal memory simulanteously. If both blocks are not in the same region another vector, ADDR_REGB, is needed to hold the forward or backward block. Likewise, ADDR_REGB has a based array named LINK_FIELDB serving the same linking function as LINK_FIELD. Additionally, the vector ADDR_REGS is needed for holding the portions of a spanned record (a record which crosses more than one ADDR_REGs). SPAN_PT is a pointer based on ADDR_REGS(255) and pointing to the next additional block.

Implementation Design

This implementation will allow no more than 64 physical regions for the inverted lists. The NEXT_ADDR_REG keeps track of all available physical regions. One of these regions is used for HEADER and is kept in internal memory until all requests are processed. Initially, there is only

one block of size 2^{10} attached to the available space list at the bottom. As requests for blocks arrive, a block sufficiently large is split. If this split block is the block of size 2^{10} the next available region is attached to the list header. New available blocks and blocks desired for satisfying requests are attached to and removed from the front of the appropriate list, as in a stack. Since each block contains four bytes for the control word, two bytes for the number of addresses included in the block, and four bytes for each address, the block size is at least ten bytes. It is impossible to start a block at the last word of a region. Therefore the purpose for setting a link to the last word of a region is to distinguish whether this link points to a list header or a free block.

When an address is desired for adding to an existing inverted list and the current address block containing this list does not have enough room for the new address, the current address block is returned to the memory pool and the next larger block is fetched. Figure 14 shows the logic of releasing and allocating a block. The largest block can only contain 254 addresses. If an inverted list has more than 254 addresses, the excess addresses are spanned to additional regions and pointers are planted to these regions. The NA field of the first region of a spanned record contains the total number of addresses in the inverted list while the NA fields of the additional regions contain the number of addresses in the individual region or

block. Therefore spanned and unspanned records can be identified by testing the NA of the blocks pointed to by the pointers associated with words.

```
ALLOCATE:PROCEDURE;
    SEARCH THE AVAILABILITY LISTS FOR A CANDIDATE BLOCK;
    IF THERE IS NO CANDIDATE BLOCK THEN STOP;
    DELETE THE CANDIDATE FROM THE AVAILABILITY LIST;
    DO FOREVER;
       IF CANDIDATE CAN NOT BE SPLIT INTO BUDDIES THEN RETURN;
       IF REQUEST SIZE IS GREATER THAN THE BUDDIES CREATED BY
             SPLITTING CANDIDATE BLOCK THEN RETURN;
       SPLIT CANDIDATE INTO BUDDIES;
       IF ONE OF BUDDIES CAN SATISFY THE REQUEST
             THEN MAKE IT AS THE NEW CANDIDATE AND PUT THE
                       OTHER ON ITS AVAILABILITY LIST;
    END;
END ALLOCATE;

RELEASE:PROCEDURE;
    DO FOREVER;
       IF CANDIDATE IS THE LARGEST BLOCK THE PUT IT ON ITS
             AVAILABILITY LIST, RETURN;
       FIND THE BUDDY OF CANDIDATE;
       IF BUDDY IS FREE AND WHOLE
             THEN DELETE BUDDY FROM THE AVAILABILITY LIST
                       AND COALESCE WITH CANDIDATE;
             ELSE RETURN;
    END;
END RELEASE;
```

Figure 14. Logic for Allocating and Releasing a Block

Major Logic Design

Both low and high level PDL of the logic used with this implementation are contained in Appendices A and B. An invocation diagram for the important modules is displayed in Figure 15.

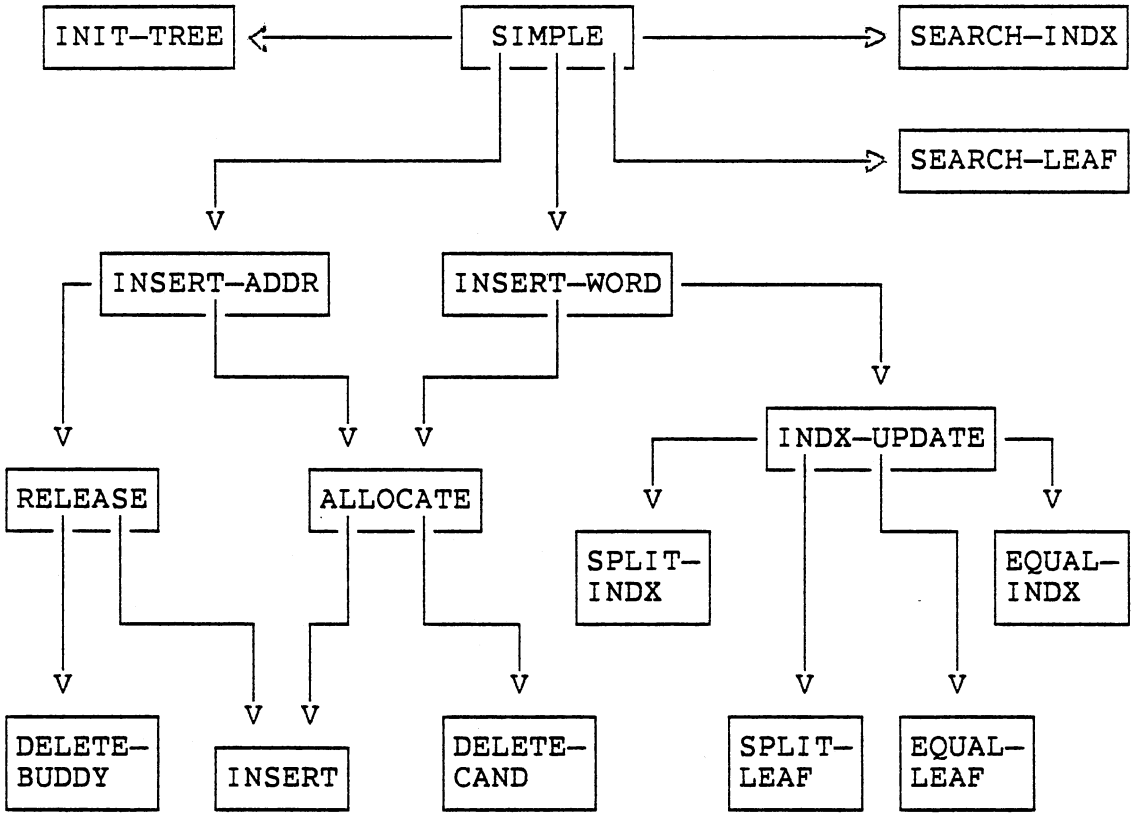


Figure 15. Diagram for Major Logic Models

The SIMPLE routine is responsible for setting up or

maintaining a simple prefix B-tree. If it is to build a new tree the INIT_TREE routine is invoked; otherwise, the SIMPLE routine reads the information concerning the existing tree into HEADER. The SEARCH_INDX and SEARCH_LEAF routines are used to find the position of the search word. If the search word is a new word the INSERT_WORD routine is invoked; otherwise, the INSERT_ADDR routine is performed. The INDX_UPDATE routine is responsible for handling overflow conditions. The ALLOCATE routine allocates a block of storage large enough to satisfy the request size. The RELEASE routine coalesces the buddies if possible and then liberates blocks of storge that have been allocated.

The block allocated for a requested size is always removed from the front of the availability list while the buddy of the released block is deleted from any location in the list. Therefore, the ALLOCATE and RELEASE routines have their own deletion routines, DELETE_CAND and DELETE_BUDDY. The INSERT routine invoked by both ALLOCATE and RELEASE attaches a free block to the front of the appropriate list.

Measure of Buddy System

The performance of buddy systems depends upon execution time and memory utilization. Since this implementation only inserts new words or addresses into the tree, recombinations are seldom encountered. The measure of execution time is the number of blocks allocated and released, instead of the number of splits and recombinations. Two arrays, ALLOC and

DEALL, are used to keep this information. ALLOC will be incremented by one when a requested block is allocated by invoking ALLOCATE. Likewise, DEALL will be incremented by one when the RELEASE routine is performed to release a block.

The measures of memory utilization are the same as in Peterson's (18) article. The INTERNAL routine traverses the simple prefix B-tree in sequential order and computes the overallocated space of the inverted lists. The EXTERNAL routine searches the available space list to find all the available blocks on the separate availability lists. The PDL of these two routines is contained in the Appendix C.

This implementation addresses memory utilization in a staged fashion. Initially, the number of available physical regions is set to ten. When these regions have been depleted, SIMPLE invokes both INTERNAL and EXTERNAL routines to compute and report the internal and external fragmentation, then increases the available regions by increments of ten. The fragmentation-measuring procedures execute recursively until all the test data is processed. Finally, SIMPLE reports the total number of blocks allocated and released.

For a comparison of the buddy systems' performances, seven different size tables, shown in Appendix D, are used. Four are generalized Fibonacci buddy sequences, namely GF-1, GF-2, GF-3, and GF-4. The others are the binary, Fibonacci, and weighted buddy systems. The test data contains fourteen

bytes for each word, two bytes for the document number, and five bytes for the word number within the document. It is sorted on word number to scramble the word order for the purpose of random insertion.

CHAPTER V

SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FURTHER WORK

Among the several B-tree implementation techniques, the Prefix B-Tree indices provide enhanced performance while indexing a large database with variable length keys. The buddy system is used as the alternate treatment of the variable-length inverted lists so as to increase storage utilization. A summary of the experimental results, advantages, disadvantages, practicability, and suggestions for further research of this implementation follows.

Summary of Prefix B-Tree Indices

Bayer and Unterauer (1) have conducted an implementation comparing the performance of B^+-trees, simple prefix B-trees, and prefix B-trees. The following main results are obtained (1, p.24). First, the time to perform the algorithms for simple prefix B-trees is nearly identical to the time for B^+-trees, while prefix B-trees need 50-100 percent more time. Second, there is no difference in the number of disc accesses when the trees have less than 200 pages. For trees having between 400 and 800 pages, simple prefix B-trees require 20-25 percent fewer disc accesses

than B^+-trees. Prefix B-trees need about 2 percent fewer disc accesses than simple prefix B-trees.

Compared to B^+-trees, the Prefix B-Tree indices have the advantages of less storage requirements for the index part and fast retrieval time from secondary storage, but have the disadvantages of more complex index-building and maintenance algorithms and much higher computing times. For indexing large textual databases in which the words are variable in length, occur in clusters, and reside on secondary storage for external searching, Prefix B-Tree indices are very suitable.

Summary of Dynamic Lists

The purpose of buddy systems is to keep track of the common memory pools used to satisfy storage requires. Two aspects of buddy systems are important: execution time and memory utilization. The experimental results are classified into four categories for performance analysis, namely internal fragmentation, external fragmentation, total fragmentation, and execution time.

These results, presented in Tables III - VI, are almost identical to the results, mentioned in page 30, in Peterson's (18) article. The following general conclusions can be made from the Table III - VI. The weighted buddy system has the best performance among all the systems in terms of internal fragmentation, but can be recommended only when the distribution of block sizes is skewed towards small

TABLE III

PERCENTAGE INTERNAL FRAGMENTATION
FOR BUDDY SYSTEMS

Systems	Regions Used				
	10	20	30	40	50
Binary	32.36	33.18	31.84	30.75	31.28
Weighted	26.43	24.36	22.96	21.79	21.66
Fibonacci	44.69	39.28	37.64	36.09	34.50
GF-1	39.77	35.96	34.24	32.06	32.39
GF-2	38.02	33.59	33.15	32.06	29.37
GF-3	43.06	38.43	37.81	36.45	34.10
GF-4	29.50	27.07	26.61	27.05	25.01

TABLE IV

PERCENTAGE EXTERNAL FRAGMENTATION
FOR BUDDY SYSTEMS

Systems	Regions Used				
	10	20	30	40	50
Binary	0.78	0.00	1.30	0.19	3.09
Weighted	22.65	18.12	17.53	15.74	15.00
Fibonacci	2.42	0.93	1.64	1.17	3.02
GF-1	1.25	1.99	0.36	0.00	0.75
GF-2	0.00	0.23	4.74	6.58	1.29
GF-3	0.00	0.00	3.12	4.64	2.10
GF-4	2.96	1.25	4.01	9.49	0.90

TABLE V

PERCENTAGE TOTAL FRAGMENTATION
FOR BUDDY SYSTEMS

Systems	Regions Used				
	10	20	30	40	50
Binary	32.88	33.18	32.72	30.88	33.40
Weighted	43.09	38.06	36.36	34.10	33.41
Fibonacci	46.03	39.84	38.66	37.19	36.07
GF-1	40.52	37.23	34.47	32.06	32.89
GF-2	38.02	33.74	36.31	36.53	30.83
GF-3	43.06	38.43	39.75	39.39	35.48
GF-4	31.58	27.98	29.54	33.97	25.68

TABLE VI

TOTAL NUMBER OF ALLOCATED AND RELEASED
BLOCKS FOR BUDDY SYSTEMS

Systems	Allocated	Released
Binary	2611	977
Weighted	3146	1512
Fibonacci	2323	1289
GF-1	2564	920
GF-2	2580	946
GF-3	2507	873
GF-4	2740	1106

block sizes. In most cases it can not be recommended because its execution time and external fragmentation are worse than those of any other system. The performance of Fibonacci system is worse in this implementation than in Peterson's implementation. No ideal Fibonacci sequence can be found to fit this physical region size with a fullword alignment of block sizes.

For a real request distribution, fragmentations may be considerably different depending upon the "fit" of the provided block sizes to the requested block sizes. Obviously, GF-4 fits quite closely to the storage requirements and has the best performance among all the systems. If the information concerning the distribution of requests that will be serviced by the memory management system is absent, the binary buddy system is recommended due to its average performance. However, when statistics are available on the distribution of requests, the generalized Fibonacci buddy system proposed by Burton (3) is a good buddy system that can be tailored to any storage allocation requirement.

Conclusions and Suggested Further Work

If the inverted lists are stored immediately after the words B-tree schemes only guarantee 50 percent storage utilization. Furthermore, a significant portion of the leaf node may be used to store the inverted lists, instead of the words, so as to increase the tree levels and decrease the

storage utilization. Compared to B-tree schemes, the buddy systems used to manage the separate space for variable length inverted listes can save a considerable amount of space and simplify the inverted lists' management. These buddy systems are worthwile to implement, especially for relatively large textual databases when there is knowledge of the storage request distribution. However, some futher studies still can be made.

As Bayer and Unterauer (1) suggested, choosing a suitable split point within a split interval can reduce the length of shortest separators and save more space for the index nodes. But, unfortunately, this technique might lead to worse storage utilization because the split point might not be in the middle of the splitting node, therefore causing some nodes to be less than half full. More experimental implementations can be made to find the efficiency, in terms of height and the average storage utilization, in split intervals.

Furthermore, it is necessary to perform empirical studies to provide adequate block sizes and efficient region sizes for actual use. Further experimentation should be conducted to determine the fewest number of regions required to minimize the time and space costs. Such investigation is very useful in determining the practicality of combining a B-tree scheme and a buddy system for future use.

BIBLIOGRAPHY

(1) Bayer, R. and Unterauer, K. "Prefix B-Tree." ACM Transaction Database System, Vol. 2 (March, 1977), 11-16.

(2) Bogart, T. G. "An Experimental System for Dynamically Managing Secondary Storage." (Unpub. M.S. Report, Oklahoma State University, 1978.)

(3) Burton, Warren. "A Buddy System Variation for Disk Storage Allocation." Comm. ACM, Vol 19, No. 7 (July, 1976), 416-417.

(4) Chang, H. K. "Commpressed Indexing Method." IBM Technical Disclosure Bulletin, Vol. 11, No. 11 (April, 1969), 1400.

(5) Christian, D. D. "A B-Tree Index Approach to Storing and Retrieving Records on Direct Access Auxiliary Storage." (Unpub. M.S. Thesis, Oklahoma State University, 1977.)

(6) Comer, D. "The Ubiquitous B-Tree." Computing Surveys, Vol. 11, No. 2 (June, 1979), 121-137.

(7) Cranston, B. and Thomas, R. "A Simplified Recombination Scheme for the Fibonacci Buddy System." Comm. ACM, Vol. 18, No. 6 (June, 1975), 331-332.

(8) Dasananda, Surapol. "Fibonacci-Based Buddy Systems." (Unpub. M.S. report, Oklahoma State University, 1976.)

(9) Feng, A. L. "A Study of Two Competing Index Mechanisms: Prefix B^+_Tree and Trie structures." (Unpub. M.S. Thesis, Oklahoma State University, 1982.)

(10) Held, G. and Stonebraker, M. "B-Tree Re-examined." Comm. ACM, Vol. 21, No. 2 (Feb., 1978), 139-143.

(11) Hinds, J. A. "An Algorithm for Locating Adjacent Storage Blocks in the Buddy System." Comm. ACM, Vol. 18, No. 4 (April, 1975), 221-222.

(12) Hirschberg, D. S. "A Class of Dynamic Memory Allocation Algorithms." Comm. ACM, Vol. 16, No. 10 (Oct., 1973), 615-618.

(13) Keehn, D. G. and Lacy, J. O. "VSAM Data Set Design Parameters." IBM Syst. J., Vol. 13, No. 3 (1974), 186-213.

(14) Knowlton, K. C. "A Fast Storage Allocator." Comm. ACM, Vol. 8, No. 10 (Oct., 1965), 623-625.

(15) Knuth, D. E. The Art of Computer Programming Vol. 1: Fundamental Algorithms, Addison Wesley Publ. Co., Reading Mass., 1973.

(16) Knuth, D. E. The Art of Computer Programming Vol. 3: Sorting and Searching, Addison Wesley Publ. Co., Reading Mass., 1973.

(17) OS PL/I Checkout and Optimizing Compilers: Language Reference Manual (GC33-0009-4). New York: International Business Machines Corporation, 1976.

(18) Peterson, J. L. "Buddy Systems." Comm. ACM, Vol. 20, No. 6 (June 1977), 421-431.

(19) Shen, K. K. and Peterson, J. L. "A Weighted Buddy Method for Dynamic Storage Allocation." Comm. ACM, Vol. 17, No. 10 (Oct. 1974), 558-562.

(20) Wagner, R. E. "Indexing Design Considerations." IBM Syst. J., Vol. 12, No. 4(1973), 351-367.

(21) Webster, R. E. "B$^+$-Tree." (Unpub. M.S. Report, Oklahoma State University, 1980.)

APPENDIX A

HIGH LEVEL PDL FOR PREFIB B-TREE STRUCTURE

AND ASSOCIATED DYNAMIC LISTS

```
SIMPLE:PROCEDURE(MAIN);
    READ OPTIONS CARD;
    IF THIS RUN IS A NEW TREE THEN CALL INIT_TREE;
    ELSE READ ADDRESS REG(0) INTO HEADER;
    READ A INSERT WORD;
    DO WHILE(MORE INSERT WORDS);
       SET WORD FOUND = FALSE;
       SET THE STARTING SEARCH POINT = ROOT NODE;
       DO WHILE(INDEX PART);
          CALL SEARCH_INDX;
          PUT THE SEARCH PATH ON STACK;
       END;
       IF TREE IS NOT EMPTY THEN CALL SEARCH_LEAF;
       IF WORD FOUND THEN CALL INSERT_ADDR;
                     ELSE CALL INSERT_WORD;
       READ NEXT INSERT WORD;
    END;
    WRITE HEADER;
END SIMPLE;


INIT_TREE:PROCEDURE;
    READ IN THE SIZE NUMBER;
    IF THE SIZE NUMBER EXCEEDS THE MAXINUM THEN PRINT ERROR;
    INITIALIZE THE TREE EMPTY;
    READ IN THE TABLE OF SIZES;
    INITIALIZE THE AVAILABILITY LISTS EMPTY;
    PUT ADDRESS REG(1) ON THE PROPER LIST;
    PERFORMAT THE FILES;
    INITIALIZE THE CONTROL FIELD FOR ADDRES REG(1);
    INITIALIZE THE LINK FIELD FOR ADDRES REG(1);
    WRITE ADDRESS REG(1);
END INIT_TREE;


SEARCH_INDX:PROCEDURE;
    READ INDEX REGION;
    SET DISPLACEMENT = THE POSITION OF THE FIRST SEPARATOR;
```

```
    DO N = 1 TO NUMBER OF SEPARATOR;
       IF INSERT WORD < SEPARATOR
          THEN DO;
               SET REGION NUMBER = LEFT POINTER;
               SET RIGHT BROTHER REGION = RIGHT POINTER;
               SET RIGHT SEPARATOR = SEPARATOR;
               RETURN;
          END;
       ELSE DO;
            SET LEFT BROTHER REGION = LEFT POINTER;
            SET LEFT SEPARATOR = SEPARATOR;
            INCREMENT DISPLACEMENT TO THE POSITION OF THE
                      NEXT SEPARATOR;
       END;
    END;
    SET DISPLACEMENT TO THE FIRST UNUSED BYTE;
    SET REGION NUMBER = RIGHT POINTER;
END SEARCH_INDX;


SEARCH_LEAF:PROCEDURE;
    READ LEAF REGION;
    SET DISPLACEMENT = THE POSITION OF THE FIRST WORD;
    DO N = 1 TO NUMBER OF WORD;
       IF INSERT WORD = WORD;
          THEN DO;
               SET WORD FOUND = TRUE;
               SET ADDRESS BLOCK LOCATION = RIGHT POINTER;
               RETURN;
          END;
       ELSE IF INSERT WORD > WORD
               THEN INCREMENT DISPLACEMENT TO THE POSITION
                    OF THE NEXT WORD;
       ELSE RETURN;
    END;
END SEARCH_LEAF;


INSERT_ADDR:PROCEDURE;
    READ ADDRESS REGION POINTED BY LEAF NODE;
    IF IT IS A SPANNED RECORD THEN
          FIND THE LAST ADDRESS BLOCK OF THIS SPANND RECORD;
    MOVE ADDRESS BLOCK TO TEMP ADDRESS BLOCK;
    READ NEW ADDRESS INTO TEMP ADDRESS BLOCK;
    COMPUTE REQUEST SIZE;
    IF REQUEST SIZE > ORIGINAL ADDRESS BLOCK SIZE
       THEN DO;
            CALL RELEASE;
            CALL ALLOCATE;
            IF IT IS NOT A SPANNED RECORD THEN UPDATE THE
               POINTER ASSOCIATED WITH THE INSERT WORD
               IN THE LEAF NODE;
```

```
      END;
   MOVE TEMP ADDRESS BLOCK TO THE AVAILABLE BLOCK;
   DO WHILE(MORE ADDRESSES);
      READ IN ADDRESSES TO TEMP ADDRESS BLOCK;
      COMPUTE REQUEST SIZE;
      CALL ALLOCATE;
      MOVE TEMP ADDRESS BLOCK TO THE AVAILABLE BLOCK;
      UPDATE THE POINTER ON THE SPANNED REGION;
      WRITE SPANNED REGION;
   END;
   WRITE THE AVAILABLE REGION;
   UPDATE TOTAL NUMBER OF ADDRESSES;
END INSERT_ADDR;


INSERT_WORD:PROCEDURE;
   READ IN ADDRESSES TO TEMP ADDRESS BLOCK;
   COMPUTE REQUEST SIZE;
   CALL ALLOCATE;
   MOVE TEMP ADDRESS BLOCK TO THE AVAILABLE BLOCK;
   PUT THE INSERT WORD AND ADDRESS BLOCK LOCATION INTO
         LEAF NODE ON PROPER POSITION;
   IF TOO MANY WORDS TO FIT ON ONE LEAF NODE
         THEN CALL INDX_UPDATE;
         ELSE WRITE LEAF NODE;
   DO WHILE(MORE ADDRESSES);
      READ IN ADDRESSES TO TEMP ADDRESS BLOCK;
      COMPUTE REQUEST SIZE;
      CALL ALLOCATE;
      MOVE TEMP ADDRESS BLOCK TO THE AVAILABLE BLOCK;
      UPDATE THE POINTER ON THE SPANNED REGION;
      WRITE SPANNED REGION;
   END;
   WRITE THE AVAILABLE REGION;
   UPDATE THE TOTAL NUMBER OF ADDRESSES;
END INSERT_WORD;


ALLOCATE:PROCEDURE;
   IF REQUEST SIZE > REGION SIZE THEN PRINT ERROR;
   SEARCH AVAILABILITY SPACE LIST FOR A CANDIDATE BLOCK;
   IF THERE IS NOT A BLOCK LARGE ENOUGH THEN PRINT ERROR;
   STARTING LOCATION = ADDRESS OF CANDIDATE;
   READ CANDIDATE REGION INTO ADDRESS REGION;
   CALL DELETE_CAND;
   DO FOREVER;
      MARK CANDIDATE USED;
      IF THE CANDIDATE CAN NOT BE SPLIT INTO BUDDIES
                                          THEN RETURN;
      IF REQUEST SIZE > THE BUDDIES CREATED BY SPLITTING
                              THE CANDIDATE THEN RETURN;
      SPLIT THE CANDIDATE INTO LEFT AND RIGHT BUDDY;
```

```
        SET CONTROL FIELDS FOR BOTH BUDDIES;
        IF REQUEST SIZE > RIGHT BUDDY
           THEN DO;
                MARK RIGHT BUDDY FREE;
                MARK LEFT BUDDY USED;
                CALL INSERT(RIGHT BUDDY, LIST);
                MARK LEFT BUDDY THE NEW CANDIDATE BLOCK;
           END;
        ELSE DO;
             MARK RIGHT BUDDY USED;
             MARK LEFT BUDDY FREE;
             CALL INSERT(LEFT BUDDY, LIST);
             MARK RIGHT BUDDY THE NEW CANDIDATE BLOCK;
        END;
     END;
END ALLOCATE;


DELETE_CAND:PROCEDURE;
     INCREMENT DISPLACEMENT BY ONE;
     SET RLINK OF THE LIST = RLINK OF CANDIDATE;
     IF ONLY ONE BLOCK ON THE LIST
        THEN DO;
          IF CANDIDATE SIZE = REGION SIZE
             THEN DO;
               SET THE LINKS FOR NEXT ADDRESS REGION = LIST;
               SET THE LINKS FOR THE LIST =
                                THE NEXT ADDRESS REGION;
               INCREMENT NEXT ADDRESS REGION BY ONE;
             END;
          ELSE SET LLINK OF THE LIST = RLINK OF THE LIST;
          DECREMENT DISPLACEMENT BY ONE;
          RETURN;
        END;
     IF CANDIDATE AND RLINK OF CANDIDATE ARE IN THE SAME
        REGION THEN DO;
          SET LLINK(RLINK OF CANDIDATE) = LLINK OF CANDIDATE;
          DECREMENT DISPLACEMENT BY ONE;
          RETURN;
        END;
     READ RLINK OF CANDIDATE INTO ADDR_REGB;
     SET LLINK(RLINK OF CANDIDATE) = LLINK OF CANDIDATE;
     WRITE RLINK OF CANDIDATE FROM ADDR_REGB;
     DECREMENT DISPLACEMENT BY ONE;
END DELETE_CAND;


RELEASE:PROCEDURE;
     DO FOREVER;
        IF THE CANDIDATE IS THE LARGEST ALLOWABLE SPACE
           THEN DO;
                MARK IT FREE;
```

```
                CALL INSERT(CANDIDATE LOC, LIST);
                WRITE CANDIDATE REGION FROM ADDR_REG;
                RETURN;
             END;
          FIND THE ADDRESS OF THE CANDIDATE'S BUDDY;
          IF (BUDDY IS USED) OR (BUDDY IS SPLIT)
             THEN DO;
                MARK CANDIDATE FREE;
                CALL INSERT(CANDIDATE LOC, LIST);
                WRITE CANDIDATE REGION FROM ADDR_REG;
                RETURN;
             END;
          CALL DELETE_BUDDY;
          IF BUDDY ADDRESS < CANDIDATE ADDRESS
                           THEN MARK BUDDY THE NEW CANDIDATE;
          COALESCE THE BUDDIES;
       END;
END RELEASE;


DELETE_BUDDY:PROCEDURE;
       INCREMENT DISPLACEMENT BY ONE;
       IF ONLY ONE BLOCK ON THE LIST
          THEN DO;
                LINKS OF LIST = RLINK OF BUDDY;
                DECREMENT DISPLACEMENT BY ONE;
                RETURN;
          END;
       IF LLINK OF THE BUDDY POINTS TO A LIST HEADER
          THEN RLINK OF THE LIST = RLINK OF BUDDY;
          ELSE DO;
                IF BUDDY AND LLINK OF BUDDY ARE IN THE SAME
                   REGION THEN RLINK(LLINK OF BUDDY) =
                                                RLINK OF BUDDY;
                ELSE DO;
                     READ LLINK OF BUDDY INTO ADDR_REGB;
                     SET RLINK(LLINK OF BUDDY) = RLINK OF BUDDY;
                     WRITE LLINK OF BUDDY FROM ADDR_REGB;
                END;
          END;
       IF RLINK OF THE BUDDY POINTERS TO A LIST HEADER
          THEN LLINK OF LIST = LLINK OF BUDDY;
          ELSE DO;
                IF BUDDY AND RLINK OF BUDDY ARE IN THE SAME
                   REGION THEN LLINK(RLINK OF BUDDY) =
                                                LLINK OF BUDDY;
                ELSE DO;
                     READ RLINK OF BUDDY INTO ADDR_REGB;
                     SET LLINK(RLINK OF BUDDY) = LLINK OF BUDDY;
                     WRITE RLINK OF BUDDY FROM ADDR_REGB;
                END;
          END;
       DECREMENT DISPLACEMENT BY ONE;
```

```
END DELETE_BUDDY;


INSERT:PROCEDURE(BUDDY LOC, LIST);
    INCREMENT DISPLACEMENT BY ONE;
    IF THE LIST IS EMPTY
       THEN DO;
            SET THE LINKS FOR THE BUDDY = LIST;
            SET THE LINKS FOR THE LIST = BUDDY;
            DECREMENT DISPLACEMENT BY ONE;
            RETURN;
       END;
    SET LLINK OF BUDDY = LIST;
    SET RLINK OF BUDDY = RLINK OF THE LIST;
    IF BUDDY AND RLINK OF LIST ARE NOT IN THE SAME REGION
       THEN DO;
            READ RLINK OF LIST INTO ADDR_REGB;
            LLINK(RLINK OF LIST) = BUDDY;
            RLINK OF LIST = BUDDY;
            WRITE RLINK OF LIST FROM ADDR_REGB;
            DECREMENT DISPLACEMENT BY ONE;
            RETURN;
       END;
    LLINK(RLINK OF LIST), RLINK OF LIST = BUDDY;
    DECREMENT DISPLACEMENT BY ONE;
END INSERT;


INDX_UPDATE:PROCEDURE;
    ONLY ONE LEVEL THEN DO;
                        CALL SPLIT_LEAF;
                        CREATE A NEW ROOT;
                        RETURN;
                    END;
    READ PARENT OF LEAF REGION INTO P_REG;
    IF LEAF REGION IS NOT THE RIGHTMOST BROTHER
       THEN DO;
            READ RIGHT BROTHER INTO B_REG;
            CALL EQUAL_LEAF(TEMP_REG,B_REG,RIGHT);
       END;
    IF LEAF REGION IS NOT THE LEAFMOST BROTHER AND
       RIGHT EQUALIZATION FAIL THEN DO;
            READ LEAF BROTHER INTO B_REG;
            CALL EQUAL_LEAF(B_REG,TEMP_REG,LEFT);
       END;
    IF EQUALIZATION SUCCEEDS THEN RETURN;
    CALL SPLIT_LEAF;
    DO WHILE(INDEX PART);
       PUT PROPAGATED SEPARATOR AND PARENT REGION INTO
            TEMP_REG ON PROPER POSITION;
       IF USED BYTES IN TEMP_REG ≤ REGION SIZE
          THEN DO;
```

```
                SET I_REG = TEMP_REG;
                WRITE INDEX REGION FROM I_REG;
                RETURN;
            END;
         READ PARENT OF P_REG INTO P_REG;
         IF INDEX REGION IS NOT THE RIGHTMOST BROTHER
            THEN DO;
                READ RIGHT BORTHER INTO B_REG;
                CALL EQUAL_INDX(TEMP_REG,B_REG,RIGHT);
            END;
         IF INDEX REGION IS NOT LEFTMOST BROTHER AND
            RIGHT EQUALIZATION FAIL THEN DO;
                READ LEFT BROTHER INTO B_REG;
                CALL EQUAL_INDX(B_REG,TEMP_REG,LEFT);
            END;
         IF EQUALIZATION SUCCEEDS THEN RETURN;
         CALL SPLIT_INDX;
      END;
      CREATE A NEW ROOT;
END INDX_UPDATE;


SPLIT_LEAF:PROCEDURE;
      FIND THE SPLIT POINT WITHIN A CERTAIN SPLIT INTERVAL;
      SPLIT THE TEMP_REG INTO L_REG AND B_REG;
      SET LEAF HEADERS FOR BOTH L_REG AND B_REG;
      WRITE LEAF REGION FROM L_REG;
      WRITE LEAF REGION FROM B_REG;
      COMPUTE THE SHORTEST SEPARATOR BETWEEN L_REG AND B_REG;
END SPLIT_LEAF;


SPLIT_INDX:PROCEDURE;
      FIND THE SPLIT POINT WITHIN A CERTAIN SPLIT INTERVAL;
      SET THE PROPAGATED SEPARATOR = THE SEPARTOR ON THE
                                                 SPLIT POINT;
      SPLIT THE TEMP_REG INTO I_REG AND B_REG;
      SET INDEX HEADERS FOR BOTH I_REG AND B_REG;
      WRITE INDEX REGION FROM I_REG;
      WRITE INDEX REGION FROM B_REG;
END SPLIT_INDX;


EQUAL_LEAF:PROCEDURE(FRONT,REAR,DIRECT);
      IF THE TOTAL LENGTH OF FRONT AND REAR > 2048
         THEN RETURN;
      COMBINE FRONT AND REAR INTO TOTAL REGION;
      FIND THE SPLIT POINT OF TOTAL REGION;
      IF DIRECT = RIGHT
         THEN SPLIT TOTAL REGION INTO L_REG AND B_REG;
         ELSE SPLIT TOTAL REGION INTO B_REG AND L_REG;
```

```
    COMPUTE THE SHORTEST SEOARATOR BETWEEN L_REG AND B_REG;
    SET LEAF HEADERS FOR BOTH L_REG AND B_REG;
    WRITE LEAF REGION FROM L_REG;
    WRITE LEAF REGION FROM B_REG;
    UPADTE THE SHORTEST SEPARATOR;
    WRITE PARENT REGION FROM P_REG;
END EQUAL_LEAF;


EQUAL_INDX:PROCEDURE(FRONT,REAR,DIRECT);
    IF THE TOTAL LENGTH OF FRONT AND REAR > 2048
       THEN RETURN;
    COMBINE FRONT, PROPAGATED SEPARATOR FOR FRONT AND REAR,
       AND REAR INTO TOTAL REGION;
    FIND THE SPLIT POINT OF TOTAL REGION;
    SET PROPAGATED SEPARATOR = SEPARATOR ON THE SPLIT POINT;
    IF DIRECT = RIGHT
       THEN SPLIT TOTAL REGION INTO I_REG AND B_REG;
       ELSE SPLIT TOTAL REGION INTO B_REG AND I_REG;
    SET INDEX HEADERS FOR BOTH I_REG AND B_REG;
    WRITE INDEX REGION FROM I_REG;
    WRITE INDEX REGION FROM B_REG;
    UPADTE THE PROPAGATED SEPARATOR;
    WRITE PARENT REGION FROM P_REG;
END EQUAL_INDX;
```

APPENDIX B

LOW LEVEL PDL DESCRIPTION FOR PREFIX B-TREE AND DYNAMIC LISTS

Description of Variables for Low Level PDL

TREE - REGIONAL(1) file that contains the available space for the index of words.

ADRS - REGIONAL(1) file that contains the available space for inverted lists.

TBSZ - sequential file that contains the size table.

INFL - sequential file that contains words and addresses.

I_REG - vector of bytes that represents an index node.

L_REG - vector of bytes that represents a leaf node.

B_REG - vector of bytes that is the brother of a full node.

P_REG - vector of bytes that is the parent of a full node.

ADDR_REG - vector of words that contains inverted lists.

ADDR_REGB - vector of words that is the brother of ADDR_REG.

ADDR_REGS - vector of words that contains the portion of a spanned record.

TOTAL_REG - vector of bytes that contains a full node and its brother node.

TEMP_REG - vector of bytes that contains a full node and the inserted key or propagated separator.

HEADER - the first region of ADRS. It contains the information of a tree. It has ten elementary items.
ROOT - root node of the tree.
LEVEL - number of levels of the tree.

NEXT_TREE_REG - next available region in TREE.
NEXT_ADDR_REG - next available region in ADRS.
SIZE_NUM - number of block sizes.
SIZE_TABLE - structure that contains the block sizes, Fibonacci factor, and the headers of available lists. It has four elementary items.
SIZE - the size of a block
K - the Fibonacci factor.
SLLINK - the left link used for linkage to availability lists. It contains region number, SLREG, and offset, SLDISP.
SRLINK - the right link used for linkage to availability lists. It contains region number, SRREG, and offset, SRDISP.

CONTROL_FIELD - structure that contains all information about a block. It has four elementary items and is based on ADDR_REG(0).
IFIELD - represents the I-field in Burton's (3) article. See Chapter III.
FREE,LBC,ISIZE - represent FREE, LBC, and SIZE in Hinds' (11) article. see Chapter VI.

LINK_FIELD - structure that contains the link fields for the linked lists. It has two elementary items, LLINK and RLINK, and is based on ADDR_REG(0).

LINK_FIELDB - same as LINK_FIELD but based on ADDR_REGB(0).

SPAN_POINT - structure that is a pointer and points to additional address block. It has two elementary items, AREG_NUM and AREG_DISP, is based on ADDR_REGS(255).

ADDR_FIELD - structure that contains the information of a inverted list. It has two elementary items, NUM_ADDR and ADDRESS, is dynamically based on ADDR_REG(AREG_DISP+1).

I_HEAD - structure that contains the information about an index node. It has two items as follows and is based on I_REG(1).
I_UNUSE_BYTE - unused bytes in index node.
NUM_PW - number of separators in the index node.

L_HEAD - structure that contains the information about a leaf node. It has three items as follows and is based on L_REG(1).
HORIZONTAL - Horizontal pointer points to the next leaf node in order.
L_UNUSE_BYTE - unused bytes in the leaf node.
NUM_WORD - number of words in the leaf node.

P_HEAD - same as I_HEAD but based on P_REG(1).

BI_HEAD - same as I_HEAD but based on B_REG(1).

BL_HEAD - same as L_HEAD but based on B_REG(1);

I_INFO - structure that contains the information about a separator. It has three items and is dynamically based on I_REG(IREG_DISP).
LPOINT - Pointer points to the left descendant.
PW_LEN - length of separator.
PART_WORD - separator

RPOINT - Pointer points to the right descendant and is dynamically based on I_REG(IREG_DISP+2+PW_LEN).

L_INFO - structure that contains the information about a word. It has two items and is dynamically based on L_REG(LREG_DISP).
LPOINT - Pointer points to left descendant.
WORD_LEN - length of word.
WRD - word

APOINT - Pointer points to a inverted list and is dynamically based on L_REG(LREG_DISP+2+WORD_LEN).

IREG_NUM,LREG_NUM,BREG_NUM,PREG_NUM,AREG_NUM,AREG_NUMB,
AREG_NUMS - the region number of an index node, leaf node, brother node, parent node, address region, buddy region, and spanned region, respectively.

IREG_DISP,LREG_DISP,BREG_DISP,PREG_DISP,AREG_DISP,
ADDR_DISPS - offset within the index node, leaf node, brother node, parent node, address region, and spanned region, respectively.

TEMP_ADDR_BLK - vector of words that contains the input addresses.

MORE_WORD - indicates if there is more words to be inserted.

BUDDY FLAG - indicates if the buddy is whole, or subdivided.

WORD_FOUND - indicates if the inserted word exists.

TREE_STATUS - 'NEW' starts from a new tree.

EQUAL - If equalization succeeds.

FINISH - indicates if the input addresses belongs to the same word.

FIRST_WORD - the first word of the right leaf node.

LAST_WORD - the last word of the left leaf node.

PROP_SS - propagated separator.

SS - separator that separates two leaf nodes.

WRDADDR - structure that has three items as follows.
INS_WORD - the word to be inserted.
CDOCN - document number where the INS_WORD exists.
CWRDN - word number within CDOCN.

START_PT - the starting point to put addresses into TEMP_ADDR_BLK.

END_PT - the ending point of the last address in the TEMP_ADDR_BLK.

REQ_SIZE - a size for an allocation.

PATH - structure that contains the information about search path. It has six items.
PATH_REG - region number of the region that has been searched.
PATH_DISP - offset within the searched region.
PATH_L_BRO - region number of the left brother of the searched region.
PATH_R_BRO - region number of the right brother of the searched region.
PATH_L_WORD - the separator in the left handside of PATH_DISP.
PATH_R_WORD - the separator in the right handside of PATH_DISP.

Low Level PDL Description

```
SIMPLE:PROCEDURE(MAIN);
   BASE I_HEAD ON I_REG(0);
   BASE L_HEAD ON L_REG(0);
   BASE BI_HEAD ON B_REG(0);
   BASE BL_HEAD ON B_REG(0);
   BASE P_HEAD ON P_REG(0);
   BASE CONTROL_FIELD ON ADDR_REG(0);
   BASE LINK_FIELD ON ADDR_REG(0);
   BASE CONTROL_FIELDB ON ADDR_REGB(0);
   BASE SPAN_POINT ON ADDR_REGS(0);
   ON ENDFILE(INFL) MORE_WORD=FALSE;
   READ OPTIONS CARD;
   IF TREE_STATUS = 'NEW' THEN CALL INIT_TREE;
   ELSE DO;
        OPEN FILE(ADRS) DIRECT UPDATE;
        OPEN FILE(TREE) DIRECT UPDATE;
        READ FILE(ADRS) INTO(HEADER) KEY(0);
   END;
   /*  INPUT THE FIRST INSERT WORD  */
   OPEN FILE(INFL) INPUT SEQUENTIAL;
   READ FILE(INFL) INTO(WRDADDR);
   DO WHILE(MORE_WORD);
        WORD_FOUND=FALSE;
        IREG_NUM=ROOT;    /* SEARCH FROM ROOT */
        L = LEVEL;
        DO WHILE( L > 1 );     /* SEARCH INDEX PART */
             PATH_REG(L) = IREG_NUM;
             CALL SEARCH_INDX;
             PATH_DISP(L) = IREG_DISP;
             L = L-1;
        END;
        IF LEVEL=0 THEN DO; /* INSERT INTO A NEW TREE */
                     NEXT_TREE_REG,LEVEL = 1;
                     READ FILE(TREE) INTO(L_REG) KEY(0);
                     LREG_NUM = 0;
                     LREG_DISP= 7;
                     L_UNUSE_BYTE = 1018;
                     NUM_WORD = 0;
                     HORIZONTAL = -1;
                   END;
        ELSE CALL SEARCH_LEAF;
        IF WORD_FOUND   THEN CALL INSERT_ADDR;
                        ELSE CALL INSERT_WORD;
   END;
   WRITE FILE(ADRS) FROM(HEADER) KEYFROM(0);
   CLOSE FILE(ADRS),FILE(TREE),FILE(INFL);
END SIMPLE;


INIT_TREE:PROCEDURE;
```

```
   OPEN FILE(TBSZ) INPUT;
   READ FILE(TBSZ) INTO(SIZE_NUM);
   IF SIZE_NUM > 63
       THEN DO;
            PRINT 'THE NUMBER OF SIZES EXCEEDS THE LIMIT';
            STOP;
       END;
    ROOT,LEVEL,NEXT_TREE_REG = 0;
    NEXT_ADDR_REG = 2;
    DO I = -1 TO -SIZE_NUM BY -1;
         READ FILE(TBSZ) INTO(SIZE(I));
         READ FILE(TBSZ) INTO(K(I));
         SLREG(I),SRREG(I) = I;
         SLDISP(I),SRDISP(I) = -1;
    END;
    OPEN FILE(ADRS) DIRECT OUTPUT;
    OPEN FILE(TREE) DIRECT OUTPUT;
    CLOSE FILE(ADRS),FILE(TREE);
    OPEN FILE(ADRS) DIRECT UPDATE;
    OPEN FILE(TREE) DIRECT UPDATE;
    AREG_NUM = 1;
    FREE(0) = TRUE;
    IFIELD(0) = -1;
    LBC(0) = 0;
    ISIZE(0) = -SIZE_NUM;
    LLINK(1),RLINK(1)=SLLINK(-SIZE_NUM);
    SLREG(-SIZE_NUM),SRREG(-SIZE_NUM)=1;
    SLDISP(-SIZE_NUM),SRDISP(-SIZE_NUM)=0;
    WRITE FILE(ADRS) FROM(ADDR_REG) KEYFROM(AREG_NUM);
    CLOSE FILE(TBSZ);
END INIT_TREE;


SEARCH_INDX:PROCEDURE;
   READ FILE(TREE) INTO(I_REG) KEY(IREG_NUM);
   IREG_DISP = 5;
   BASE I_INFO ON I_REG(IREG_DISP);
   PART_SS=SUBSTR(PART_WORD,1,PW_LEN);
   DO N = 1 TO NUM_PW;
      IF INS_WORD < PART_SS
         THEN DO;
              IREG_NUM=LPOINT;
              BASE RPOINT ON I_REG(IREG_DISP+4+PW_LEN);
              PATH_R_BRO(L) = RPOINT;
              PATH_R_WORD(L)= PART_SS;
              RETURN;
         END;
         ELSE DO;
              PATH_L_BRO(L) = LPOINT;
              PATH_L_WORD(L)= PART_SS;
              IREG_DISP = IREG_DISP + 4 + PW_LEN;
              BASE I_INFO ON I_REG(IREG_DISP);
              PART_SS=SUBSTR(PART_WORD,1,PW_LEN);
```

```
        END;
   END;
   IREG_DISP = IREG_DISP + 4 + PW_LEN;
   BASE RPOINR ON I_REG(IREG_DISP);
   IREG_NUM=RPOINT;
END SEARCH_INDX;


SEARCH_LEAF:PROCEDURE;
   LREG_DISP = 7;
   LREG_NUM = IREG_NUM;
   READ FILE(TREE) INTO(L_REG) KEY(LREG_NUM);
   BASE L_INFO ON L_REG(LREG_DISP);
   WORD=SUBSTR(WRD,1,WORD_LEN);
   DO N = 1 TO NUM_WORD;
      IF INS_WORD = WORD
         THEN DO;
              WORD_FOUND = TRUE;
              BASE APOINT ON L_REG(LREG_DISP+2+WORD_LEN);
              RETURN;
         END;
      ELSE IF INS_WORD > WORD
         THEN DO;
              LREG_DISP = LREG_DISP + 4 + WORD_LEN;
              BASE L_INFO ON L_REG(LREG_DISP);
              WORD=SUBSTR(WRD,1,WORD_LEN);
         END;
      ELSE RETURN;
   END;
END SEARCH_LEAF;


INSERT_ADDR:PROCEDURE;
   READ FILE(ADRS) INTO(ADDR_REG) KEY(AREG_NUM);
   FINISH = FALSE;
   BASE ADDR_FIELD ON ADDR_REG(AREG_DISP+1);
   TEMP_NUM,TOTAL_ADDR = NUM_ADDR;
   IF NUM_ADDR > 254
      THEN DO;
        ADDR_REGS = ADDR_REG;
        P_AREG_NUMS=AREG_NUM;
        DO N= 1 TO (NUM_ADDR/254-1);
           P_AREG_NUMS=AREG_NUMS;
           READ FILE(ADRS) INTO(ADDR_REGS) KEY(P_AREG_NUMS);
        END;
        READ FILE(ADRS) INTO(ADDR_REG) KEY(AREG_NUMS);
        BASE ADDR_FIELD ON ADDR_REG(AREG_DISP+1);
        AREG_NUM = AREG_NUMS;
        AREG_DISP = AREG_DISPS;
      END;
   /* MOVE ADDRESS BLOCK TO TEMP ADDRESS BLOCK */
   DO N = 1 TO NUM_ADDR;
```

```
      TEMP_ADDR_BLK(N)=ADDRESS(N);
   END;
   START_PT = NUM_ADDR + 1;
   CALL INPUT_ADDR;
   REQ_SIZE = 6 + 4*END_PT;
   IF REQ_SIZE > SIZE(ISIZE(AREG_DISP))
      THEN DO;
        CALL RELEASE;
        CALL ALLOCATE;
        IF TEMP_NUM < 254       /* NOT A SPANNED RECORD */
           THEN DO;
             REG = AREG_NUM;
             DISP= AREG_DISP;
             WRITE FILE(TREE) FROM(L_REG) KEYFROM(LREG_NUM);
           END;
        ELSE DO;        /* SPANNED RECORD */
             AREG_NUMS=AREG_NUM;
             AREG_DISPS=AREG_DISP;
             WRITE FILE(ADRS) FROM(ADDR_REGS)
                              KEYFROM(P_AREG_NUMS);
        END;
        BASE ADDR_FIELD ON ADDR_REG(AREG_DISP+1);
        NUM_ADDR=END_PT;
      END;
   /* MOVE TEMP ADDRESS BLOCK TO ADDRESS BLOCK */
   DO N = 1 TO NUM_ADDR;
      ADDRESS(N) = TEMP_ADDR_BLK(N);
   END;
   CALL REINPUT_ADDR;
   WRITE FILE(ADRS) FROM(ADDR_REG) KEYFROM(AREG_NUM);
   READ FILE(ADRS) INTO(ADDR_REG) KEY(REG);
   BASE ADDR_FIELD ON ADDR_REG(DISP+1);
   NUM_ADDR = TOTAL_ADDR;
   WRITE FILE(ADRS) FROM(ADDR_REG) KEYFROM(REG);
END INSERT_ADDR;


INSERT_WORD:PROCEDURE;
   FINISH = FALSE;
   START_PT = 1;
   TOTAL_ADDR = 0;
   CALL INPUT_ADDR;
   REQ_SIZE = 6 +4*END_PT;
   CALL ALLOCATE;
   TEMP_REG=SUBSTR(L_REG,1,LREG_DISP-1)||LENGTH(INS_WORD)||
            INS_WORD||SUBSTR(L_REG,LREG_DISP,);
   IF LENGTH(TEMP_REG) > 1024
       THEN  CALL INDX_UPDATE;
       ELSE DO;
            L_REG = TEMP_REG;
            NUM_WORD = NUM_WORD + 1;
            L_UNUSE_BYTE = 1024 - LENGTH(TEMP_REG);
            WRITE FILE(TREE) FROM(L_REG) KEYFROM(LREG_NUM);
```

```
      END;
   BASE ADDR_FIELD ON ADDR_REG(AREG_DISP+1);
   /* MOVE FROM TEMP ADDRESS BLOCK TO ADDRESS BLOCK */
   DO N = 1 TO END_PT;
      TEMP_ADDR_BLK(N)=ADDRESS(N);
   END;
   TEMP_NUM=AREG_NUM;
   CALL REINPUT_ADDR;
   WRITE FILE(ADRS) FROM(ADDR_REG) KEYFROM(AREG_NUM);
   READ FILE(ADRS) INTO(ADDR_REG) KEY(REG_NUM);
   BASE ADDR_FIELD ON ADDR_REG(AREG_DISP+1);
   NUM_ADDR = TOTAL_ADDR;
   WRITE FILE(ADRS) FROM(ADDR_REG) KEYFROM(TEMP_NUM);
END INSERT_WORD;


INPUT_ADDR:PROCEDURE;
   TEMP_WORD=INS_WORD;
   IF START_PT <= 254
      THEN DO;
        DO N = START_PT TO 254 UNTIL(FINISH=TRUE);
           TOTAL_ADDR = TOTAL_ADDR +1;
           END_PT = N;
           TEMP_ADDR_BLK(N)=BDOCN*65536+BWRDN;
           READ FILE(INFL) INTO(WRDADDR);
           IF(NOT MORE_WORD)|(INS_WORD NOT = TEMP_WORD)
                                        THEN FINISH=TRUE;
        END;
      END;
   ELSE END_PT = 254;
END INPUT_ADDR;


REINPUT_ADDR:PROCEDURE;
   DO WHILE(NOT FINISH);
      ADDR_REGS = ADDR_REG;
      TEMP_NUM = AREG_NUM;
      START_PT = 1;
      CALL INPUT_ADDR;
      REQ_SIZE = 6 + 4*END_PT;
      CALL ALLOCATE;
      AREG_NUMS = AREG_NUM;
      AREG_DISPS= AREG_DISP;
      WRITE FILE(ADRS) FROM(ADDR_REGS) KEYFROM(TEMP);
      BASE ADDR_FIELD ON ADDR_REG(AREG_DISP+1);
      NUM_ADDR=END_PT;
      DO N = 1 TO NUM_ADDR;
         ADDRESS(N) = TEMP_ADDR_BLK(N);
      END;
   END;
END REINPUT_ADDR;
```

```
ALLOCATE:PROCEDURE;
   IF REQ_SIZE > 1024
       THEN DO;
            PRINT 'REQUEST SIZE EXCEEDS THE REGION SIZE';
            STOP;
       END;
   SEARCH AVAILABILITY SPACE LIST FOR A CANDIDATE BLOCK;
   IF SEARCH FAILS
     THEN DO;
          PRINT 'NO LARGE ENOUGH BLOCK';
          STOP;
     END;
   /* THE CANDIDATE BLOCK IS ON THE LIST */
   AREG_NUM = SRREG(I);
   AREG_DISP= SRDISP(I);
   READ FILE(ADRS) INTO(ADDR_REG) KEY(AREG_NUM);
   CALL DELETE_CAND(I);
   DO WHILE ('1'B);
       FREE(AREG_DISP) = FALSE;
       IF K(I) = 0  THEN RETURN; /* CANNOT SPLIT AGAIN */
       IF REQ_SIZE > SIZE(I+K(I))
          REQ_SIZE > SIZE(I+1)  THEN RETURN;
       ISIZE(AREG_DISP) = I+1;
       LBC(AREG_DISP) = LBC(AREG_DISP)+1;
       AREG_DISPB = AREG_DISP + SIZE(I+1)/4;
       ISIZE(AREG_DISPB) = I + K(I);
       IFIELD(AREG_DISPB)=I+1;
       LBC(AREG_DISPB) = 0;
       IF REQ_SIZE > SIZE(I+K(I))
          THEN DO;
            FREE(AREG_DISPB) = TRUE;
            FREE(AREG_DISP) = FALSE;
            CALL INSERT(ISIZE(AREG_DISPB));
            I = I + 1;
          END;
       ELSE DO;
            FREE(AREG_DISPB)= FALSE;
            FREE(AREG_DISP) = TRUE;
            CALL INSERT(ISIZE(AREG_DISP));
            I = I + K(I);
            AREG_DISP = AREG_DISPB;
       END;
   END;
END ALLOCATE;


DELETE_CAND:PROCEDURE(LIST);
   AREG_DISP = AREG_DISP + 1;
   SRLINK(LIST) = RLINK(AREG_DISP);
   /* ONLY ONE BLOCK ON LIST */
   IF LDISP(AREG_DISP) = RDISP(AREG_DISP)
      LREG(AREG_DISP) = RREG(AREG_DISP)
      /* ON THE LARGEST AVAILABLE LIST */
```

```
      THEN DO;
         IF SIZE(ISIZE(AREG_DISP-1)) = 1024
           THEN DO;
              SRREG(LIST),SLREG(LIST)=NEXT_ADDR_REG;
              SRDISP(LIST),SLDISP(LIST) = 0;
              ADDR_REGB = ADDR_REG;
              FREE(0) = TRUE;
              IFIELD(0)=LIST;
              LBC(0) =0;
              ISIZE(0)= LIST;
              LREG(1),RREG(1) = LIST;
              LDISP(1),RDISP(1) = -1;
              WRITE FILE(ADRS) FROM(ADDR_REG)
                               KEYFROM(NEXT_ADDR_REG);
              NEXT_ADDR_REG = NEXT_ADDR_REG + 1;
              ADDR_REG = ADDR_REGB;
           END;
         ELSE SLLINK(LIST)=SRLINK(LIST);
         AREG_DISP=AREG_DISP-1;
         RETURN;
      END;
   /* THE CANDIDATE AND RLINK OF CANDIDATE ARE IN THE */
   /* SAME REGION                                     */
   IF FIXED(RREG(AREG_DISP)) = AREG_NUM
      THEN DO;
           LLINK(RDISP(AREG_DISP))+1)=LLINK((AREG_DISP);
           AREG_DISP = AREG_DISP - 1;
           RETURN;
      END;
   AREG_NUMB=RREG(AREG_DISP);
   READ FILE(ADRS) INTO(ADDR_REGB) KEY(AREG_NUMB);
   LLINKB(RDISP(AREG_DISP))+1)=LLINK(AREG_DISP);
   WRITE FILE(ADRS) FROM(ADDR_REGB) KEYFROM(AREG_NUMB);
   AREG_DISP = AREG_DISP - 1;
END DELETE_CAND;


RELEASE:PROCEDURE;
   DO WHILE('1'B);
     IF SIZE(ISIZE(AREG_DISP)) = 1024
        THEN DO;
          FREE(AREG_DISP) = TRUE;
          LBC(AREG_DISP) = 0;
          CALL INSERT(ISIZE(AREG_DISP));
          WRITE FILE(ADRS) FROM(ADDR_REG) KEYFROM(AREG_NUM);
                           KEYFROM(AREG_NUM);
          RETURN;
        END;
     IF LBC(AREG_DISP) = 0          /*  RIGHT BUDDY  */
        THEN DO;
          AREG_DISPB=AREG_DISP- SIZE(ISIZE(AREG_DISP))/4;
          IF AREG_DISP=SIZE(ISIZE(AREG_DISPB))+AREG_DISPB;
                       THEN BUDDY FLAG = WHOLE;
```

```
                    ·ELSE BUDDY FLAG = SPLIT;
        END;
      ELSE DO;      /*  LEFT BUDDY   */
           AREG_DISPB=AREG_DISP+SIZE(ISIZE(AREG_DISP))/4;
           IF LBC(AREG_DISP) = 0
                    THEN BUDDY FLAG = WHOLE;
                    ELSE BUDDY FLAG = SPLIT;
      END;
      IF (BUDDY IS SPLIT   BUDDY IS INUSE)
        THEN DO;
          FREE(AREG_DISP)=TRUE;
          CALL INSERT(ISIZE(AREG_DISP));
          WRITE FILE(ADRS) FROM(ADDR_REG) KEYFROM(AREG_NUM);
          RETURN;
        END;
      CALL DELETE_BUDDY;      /* COMBINE BUDDIES */
      IF AREG_DISPB < AREG_DISP THEN AGRE_DISP=AREG_DISPB;
      ISIZE(AREG_DISP)=ISIZE(AREG_DISP)-1;
      LBC(AREG_DISP)=LBC(AREG_DISP)-1;
      FREE(AREG_DISP)=TRUE;
    END;
END RELEASE;


DELETE_BUDDY:PROCEDURE;
    AREG_DISPB=AREG_DISPB+1;
    /* ONLY ONE BLOCK ON LIST */
    IF LDISP(AREG_DISPB)=RDISP(AREG_DISPB)
       LREG(AREG_DISPB)= RREG(AREG_DISPB)
       THEN DO;
            SLREG(LREG(AREG_DISPB)),
            SRREG(LREG(AREG_DISPB))=LREG(AREG_DISPB);
            SLDISP(LREG(AREG_DISPB)),
            SRDISP(LREG(AREG_DISPB))=LDISP(AREG_DISPB);
            AREG_DISPB=AREG_DISPB-1;
       END;
    /* LEFT LINK OF BUDDY POINTS TO LIST */
    IF LDISP(AREG_DISPB)=-1
       THEN SRLINK(LREG(AREG_DISPB))=RLINK(AREG_DISPB);
    ELSE DO;
       IF LREG(AREG_DISPB)=AREG_NUM
          THEN RLINK(LDISP(AREG_DISPB)+1)=RLINK(AREG_DISPB);
                                 =RLINK(AREG_DISPB);
       ELSE DO;
          READ FILE(ADRS) INTO(ADDR_REGB)
                          KEY(LREG(AREG_DISPB));
          RLINKB(LDISP(AREG_DISPB)+1)=RLINK(AREG_DISPB);
          WRITE FILE(ADRS) FROM(ADDR_REGB)
                          KEYFROM(LREG(AREG_DISPB));
       END;
    END;
    /* RIGHT LINK OF BUDDY POINTS TO LIST */
    IF RDISP(AREG_DISPB) = -1
```

```
      THEN SLLINK(RREG(AREG_DISPB))=LLINK(AREG_DISPB);
      ELSE DO;
        IF RREG(AREG_DISPB)=AREG_NUM THEN
           LLINK(RDISP(AREG_DISPB)+1)=LLINK(AREG_DISPB);
        ELSE DO;
          READ FILE(ADRS) INTO(ADDR_REGB)
                                  KEY(RREG(AREG_DISPB));
          LLINKB(RDISP(AREG_DISPB)+1)=LLINK(AREG_DISPB);
          WRITE FILE(ADRS) FROM(ADDR_REGB)
                               KEYFROM(RREG(AREG_DISPB));
        END;
      END;
   AREG_DISPB=AREG_DISPB-1;
END DELETE_BUDDY;



INSERT:PROCEDURE(LIST);
   AREG_DISP=AREG_DISP+1;
   /* LIST IS EMPTY */
   IF SLDISP(LIST)= -1   SRDISP(LIST)= -1
       THEN DO;
            LLINK(AREG_DISP),RLINK(AREG_DISP)=SRLINK(LIST);
            SLREG(LIST),SRREG(LIST)=AREG_NUM;
            SLDISP(LIST),SRDISP(LIST)=AREG_DISP;
            RETURN;
       END;
   LREG(AREG_DISP)=LIST;
   LDISP(AREG_DISP)=-1;
   RLINK(AREG_DISP)=SRLINK(LIST);
   /* THE BUDDY AND RLINK OF LIST ARE NOT IN THE SAME */
   /* REGION                                          */
   IF AREG_NUM>=SRREG(LIST)
      THEN DO;
        AREG_NUMB=SRREG(LIST);
        READ FILE(ADRS) INTO(ADDR_REGB) KEY(AREG_NUMB);
        LREGB(SRDISP(LIST)+1),SRREG(LIST)=AREG_NUM;
        LDISPB(SRDISP(LIST)+1),SRDISP(LIST)=AREG_DISP;
        WRITE FILE(ADRS) FROM(ADDR_REGB) KEYFROM(AREG_NUMB);
        RETURN;
      END;
   LREG(SRDISP(LIST)+1),SRREG(LIST)=AREG_NUM;
   LDISP(SRDISP(LIST))+1),SRDISP(LIST)=AREG_DISP;
END INSERT;



INDX_UPDATE:PROCEDURE;
   EQUAL=FALSE;
   L=L+1;
   IF LEVEL=1 THEN DO;
                   CALL SPLIT_LEAF;
                   CALL CREAT_NEW_ROOT;
                   RETURN;
```

```
            END;
PREG_NUM=PATH_REG(L);
READ FILE(TREE) INTO(P_REG) KEY(PREG_NUM);
/* LEAF REGION IS NOT THE RIGHTMOST SIBLING    */
IF PATH_R_BRO(L) >= -1
   THEN DO;
        BREG_NUM=PATH_R_BRO(L);
        READ FILE(TREE) INTO(B_REG) KEY(BREG_NUM);
        CALL EQUAL_LEAF(TEMP_REG,B_REG,RIGHT);
   END;
/* LEAF REGION IS NOT THE LEFTMOST SIBLING AND LAST */
/* EQUALIZATION IS FAIL                             */
IF PATH_L_BRO(L) >= -1   NOT EQUAL
   THEN DO;
        BREG_NUM=PATH_L_BRO(L);
        READ FILE(TREE) INTO(B_REG) KEY(BREG_NUM);
        CALL EQUAL_LEAF(B_REG,TEMP_REG,LEFT);
   END;
IF EQUAL THEN RETURN;
CALL SPLIT_LEAF;
DO WHILE(L <= LEVEL);
   TEMP_REG=SUBSTR(P_REG,1,(PATH_DISP(L)-1))    ||
            LPOINT||LENGTH(PROP_SS)||PROP_SS||RPOINT||
            SUBSTR(P_REG,PATH_DISP(L)+2);
   IF LENGTH(TEMP_REG) <= 1024
      THEN DO;
         P_REG=TEMP_REG;
         NUM_PWP=NUM_PWP+1;
         P_UNUSE_BYTE=1024-LENGTH(TEMP_REG);
         WRITE FILE(TREE) FROM(P_REG) KEYFROM(PREG_NUM);
         RETURN;
      END;
   L=L+1;
   IF L > LEVEL THEN DO;
                     CALL SPLIT_INDX;
                     CALL CREAT_NEW_ROOT;
                     RETURN;
                END;
    IREG_NUM=PREG_NUM;
    PREG_NUM=PATH_REG(L);
    READ FILE(TREE) INTO(P_REG) KEY(PREG_NUM);
    /* LEAF REGION IS NOT THE RIGHTMOST SIBLING    */
    IF PATH_R_BRO(L) >= -1
       THEN DO;
            BREG_NUM=PATH_R_BRO(L);
            READ FILE(TREE) INTO(B_REG) KEY(BREG_NUM);
            CALL EQUAL_INDX(TEMP_REG,B_REG,RIGHT);
       END;
    /*LEAF REGION IS NOT THE LEFTMOST SIBLING AND LAST*/
    /*EQUALIZATION IS FAIL                            */
    IF PATH_L_BRO(L) >= -1   NOT EQUAL
       THEN DO;
            BREG_NUM=PATH_L_BRO(L);
            READ FILE(TREE) INTO(B_REG) KEY(BREG_NUM);
```

```
                CALL EQUAL_INDX(B_REG,TEMP_REG,LEFT);
            END;
        IF EQUAL THEN RETURN;
        CALL SPLIT_INDX;
    END;
    CALL CREAT_NEW_ROOT;
END INDX_UPDATE;


SPLIT_LEAF:PROCEDURE;
    LREG_DISP=7;
    BASE L_INFO ON TEMP_REG(LREG_DISP);
    /* FIND SPLIT POINT BETWEEN 490 AND 540 */
    DO N=1 TO (NUM_WORD+1);
         IF (LREG_DISP > 490) THEN GO TO EXIT1;
         LAST_WORD=SUBSTR(WRD,1,WORD_LEN);
         LREG_DISP=LREG_DISP+4+NUM;
         BASE L_INFO ON TEMP_REG(LREG_DISP);
    END;
    EXIT1: TEMP_NUM=N-1;
    FIRST_WORD=SUBSTR(WRD,1,WORD_LEN);
    CALL COMPUT_SS(LAST_WORD,FIRST_WORD);
    PROP_SS=SS;
    SPLIT_PT=LREG_DISP;
    DO I=(TEMP_NUMW+1) TO (NUM_WORD+1);
         LAST_WORD=FIRST_WORD;
         LREG_DISP=LREG_DISP+4+WORD_LEN;
         BASE L_INFO ON TEMP_REG(LREG_DISP);
         FIRST_WORD=SUBSTR(WRD,1,WORD_LEN);
         CALL COMPUT_SS(LAST_WORD,FIRST_WORD);
         IF LENGTH(PROP_SS) >= LENGTH(SS)
                        THEN DO;
                             PROP_SS=SS;
                             SPLIT_PT=LREG_DISP;
                             TEMP_NUM=I;
                        END;
         IF LREG_DISP > 540 THEN GO TO EXIT2;
    END;
    /* SPLIT OVERFLOW LEAF REGION INTO LEAF REGION AND  */
    /* BROTHER REGION                                   */
    EXIT2: READ FILE(TREE) INTO(B_REG) KEY(NEXT_TREE_REG);
    SUBSTR(B_REG,7)=SUBSTR(TEMP_REG,SPLIT_PT);
    L_REG=SUBSTR(TEMP_REG,1,SPLIT_PT-1);
    NUM_WORDB=NUM_WORD+1-TEMP_NUM;
    NUM_WORD=TEMP_NUM;
    L_UNUSE_BYTE=1025-SPLIT_PT;
    L_UNUSE_BYTEB=1017-LENGTH(TEMP_REG)+SPLIT_PT;
    HORIZONTAL_PTB=HORIZONTAL;
    RPOINT,HORIZONTAL,BREG_NUM=NEXT_TREE_REG;
    LPOINT=LREG_DISP;
    WRITE FILE(TREE) FROM(L_REG) KEYFROM(LREG_NUM);
    WRITE FILE(TREE) FROM(B_REG) KEYFROM(BREG_NUM);
    NEXT_TREE_REG=NEXT_TREE_REG+1;
```

```
END SPLIT_LEAF;


SPLIT_INDX:PROCEDURE;
   IREG_DISP=7;
   /* FIND SPLIT POINT BETWEEN 490 AND 540 */
   DO N=1 TO (NUM_PW+1);
        IF (IREG_DISP > 490) THEN GO TO EXIT3;
        BASE I_INFO ON TEMP_REG(IREG_DISP);
        IREG_DISP=IREG_DISP+4+NUM;
   END;
   EXIT3: TEMP_NUM=N-1;
   BASE I_INFO ON TEMP_REG(IREG_DISP);
   SPLIT_PT=IREG_DISP;
   PROP_SS=SUBSTR(PART_WORD,1,PW_LEN);
   IREG_DISP=IREG_DISP+4+NUM;
   DO N=(TEMP_NUM+1) TO (NUM_PW+1);
      BASE L_INFO ON TEMP_REG(LREG_DISP);
      SS=SUBSTR(PART_WORD,1,PW_LEN);
      IF (LENGTH(PROP_SS) >= LENGTH(SS))
            THEN DO;
                 PROP_SS=SS;
                 SPLIT_PT=IREG_DISP;
                 TEMP_NUMW=N;
            END;
     IREG_DISP=IREG_DISP+4+NUM;
     IF IREG_DISP > 540 THEN GO TO EXIT4;
   END;
   /* SPLIT INTO TWO INDEX NODES */
   EXIT4: I_REG=SUBSTR(TEMP_REG,1,SPLIT_PT-1);
   LPOINT=IREG_DISP;
   RPOINT,BREG_NUM=NEXT_TREE_REG;
   NEXT_TREE_REG=NEXT_TREE_REG+1;
   READ FILE(TREE) INTO(B_REG) KEY(BREG_NUM);
   SUBSTR(B_REG,5)=SUBSTR(TEMP_REG,
                          SPLIT_PT+2+LENGTH(PROP_SS);
   NUM_PWB=NUM_PW+1-TEMP_NUM;
   NUM_PW=TEMP_NUM;
   I_UNUSE_BYTE=1025-SPLIT_PT;
   I_UNUSE_BYTEB=1021-LENGTH(TEMP_REG)+SPLIT_PT+
                                      LENGTH(PROP_SS);
   WRITE FILE(TREE) FROM(I_REG) KEYFROM(IREG_NUM);
   WRITE FILE(TREE) FROM(B_REG) KEYFROM(BREG_NUM);
END SPLIT_INDX;


CREAT_NEW_ROOT:PROCEDURE;
   ROOT=NEXT_TREE_REG;
   NEXT_TREE_REG=NEXT_TREE_REG+1;
   READ FILE(TREE) INTO(I_REG) KEY(ROOT);
   SUBSTR(I_REG,5)=LPOINT||LENGTH(PROP_SS)||PROP_SS||RPOINT;
   I_UNUSE_BYTE= 1014-LENGTH(PROP_SS);
```

```
   NUM_PW=1;
   LEVEL=LEVEL+1;
   WRITE FILE(TREE) FROM(I_REG) KEYFROM(ROOT);
END CREAT_NEW_ROOT;


EQUAL_LEAF:PROCEDURE(FRONT_REG,REAR_REG,DIRECT);
   IF (LENGTH(FRONT_REG)+LENGTH(REAR_REG))>2048 THEN RETURN;
   LREG_DISP=7;
   TOTAL_REG=FRONT_REG || SUBSTR(REAR_REG,7);
   /* FIND NEW PARTIAL SEPARATOR */
   DO N=1 TO (NUM_WORD+NUM_WORD+1);
     TEMP_NUM=TEMP_NUM+1;
     BASE L_INFO ON TOTAL_REG(LREG_DISP);
     LAST_WORD=SUBSTR(WRD,1,WORD_LEN);
     LREG_DISP=LREG_DISP+4+WORD_LEN;
     IF LREG_DISP>((LENGTH(TOTAL_REG)/2)+3) THEN GO TO OUT1;
   END;
   OUT1: BASE L_INFO ON TOTAL_REG(LREG_DISP);
   FIRST_WORD=SUBSTR(WRD,1,WORD_LEN);
   SPLIT_PT=LREG_DISP;
   IF SPLIT_PT > 1024 |
      LENGTH(TOTAL_REG)-SPLIT_PT > 1017  THEN RETURN;
   CALL COMPUT_SS(LAST_WORD,FIRST_WORD);
   IF DIRECT='RIGHT'
      THEN IF(P_UNUSE_BYTE+LENGTH(PATH_R_WORD(L))) >=
                                              LENGTH(SS)
         THEN DO;
            L_REG=SUBSTR(TOTAL_REG,1,SPLIT_PT-1);
            L_UNUSE_BYTE=1025-SPLIT_PT;
            NUM_WORD=TEMP_NUM;
            WRITE FILE(TREE) FROM(L_REG) KEYFROM(LREG_NUM);
            SUBSTR(B_REG,7)=SUBSTR(TOTAL_REG,SPLIT_PT);
            L_UNUSE_BYTEB=1017-LENGTH(TOTAL_REG)+SPLIT_PT;
            NUM_WORDB=NUM_WORD+NUM_WORDB+1-TEMP_NUM;
            WRITE FILE(TREE) FROM(B_REG) KEYFROM(BREG_NUM);
            P_REG=SUBSTR(P_REG,1,(PATH_DISP(L)+1)) ||
                  LENGTH(SS)||SS||SUBSTR(P_REG,
                  (PATH_DISP(L)+4+LENGTH(PATH_R_WORD(L))));
            P_UNUSE_BYTE=P_UNUSE_BYTE+LENGTH(
                                 PATH_R_WORD(L))-LENGTH(SS);
            WRITE FILE(TREE) FROM(P_REG) KEYFROM(PREG_NUM);
            EQUAL=TRUE;
            RETURN;
          END;
      ELSE RETURN;
   ELSE IF(P_UNUSE_BYTE+LENGTH(PATH_L_WORD(L))) >=
                                      LENGTH(SS);
      THEN DO;
           B_REG=SUBSTR(TOTAL_REG,1,SPLIT_PT-1);
           L_UNUSE_BYTEB=1025-SPLIT_PT;
           NUM_WORDB=TEMP_NUM;
           WRITE FILE(TREE) FROM(B_REG) KEYFROM(BREG_NUM);
```

```
            SUBSTR(L_REG,7)=SUBSTR(TOTAL_REG,SPLIT_PT);
            L_UNUSE_BYTE=1017-LENGTH(TOTAL_REG)+SPLIT_PT;
            NUM_WORD=NUM_WORD+NUM_WORDB+1-TEMP_NUM;
            WRITE FILE(TREE) FROM(L_REG) KEYFROM(LREG_NUM);
            P_REG=SUBSTR(P_REG,1,(PATH_DISP(L)-3-
                        LENGTH(PATH_L_WORD(L))))||LENGTH(SS)
                        ||SS||SUBSTR(P_REG,PATH_DISP(L));
            P_UNUSE_BYTE=P_UNUSE_BYTE+LENGTH(PATH_L_WORD(L))
                                                -LENGTH(SS);
            WRITE FILE(TREE) FROM(P_REG) KEYFROM(PREG_NUM);
            EQUAL=TRUE;
            RETURN;
      END;
   ELSE RETURN;
END EQUAL_LEAF;


EQUAL_INDX:PROCEDURE(FRONT_REG,REAR_REG,DIRECT);
   TEMP_NUM=NUM_PW+NUM_PWB+2;
   LREG_DISP=7;
   IF DIRECT='RIGHT'
      THEN DO;
           NUM=LENGTH(PATH_R_WORD(L));
           TOTAL_REG=FRONT_REG || NUM || PATH_R_WORD(L) ||
                     SUBSTR(REAR_REG,5);
      END;
      ELSE DO;
           NUM=LENGTH(PATH_L_WORD(L));
           TOTAL_REG=FRONT_REG || NUM || PATH_L_WORD(L) ||
                     SUBSTR(REAR_REG,5);
      END;
   /* FIND NEW PARTIAL SEPARATOR */
   DO N=1 TO TEMP_NUM/2;
       BASE I_INFO ON TOTAL_REG(IREG_DISP);
       IREG_DISP=IREG_DISP+4+NUM;
   END;
   BASE I_INFO ON TOTAL_REG(IREG_DISP);
   PROP_SS=SUBSTR(PART_WORD,1,PW_LEN);
   /* PREDETERMINE PARENT REGION */
   IF P_UNUSE_BYTE+NUM < LENGTH(ss) THEN RETURN;
   ELSE DO;
        SUBSTR(FRONT_REG,1,LREG_DISP-1)=
                           SUBSTR(TOTAL_REG,1,LREG_DISP-1);
        SUBSTR(REAR_REG,5)=
                        SUBSTR(TOTAL_REG,LREG_DISP+2+PW_LEN);
        IF (LENGTH(FRONT_REG)>1024 | LENGTH(REAR_REG)>1024)
                                                THEN RETURN;
        /* RIGHT EQUALIZATION */
        IF DIRECT='RIGHT'
           THEN DO;
             I_REG=FRONT_REG;
             SUBSTR(B_REG,5)=SUBSTR(REAR_REG,5);
             NUM_PW=TEMP_NUM/2;
```

```
                NUM_PWB=TEMP_NUM-1-NUM_PW;
                I_UNUSE_BYTE=1024-LENGTH(FRONT_REG);
                I_UNUSE_BYTEB=1024-LENGTH(REAR_REG);
                SUBSTR(P_REG,(PATH_DISP(L)+2))=
                        LENGTH(PROP_SS)||PROP_SS||
                        SUBSTR(P_REG,(PATH_DISP(L)-2-NUM));
                P_UNUSE_BYTE=P_UNUSE_BYTE-LENGTH(PROP_SS)+NUM;
             END;
             ELSE DO;      /* LEFT EQUALIZATION */
                B_REG=FRONT_REG;
                SUBSTR(I_REG,5)=SUBSTR(REAR_REG,5);
                NUM_PWB=TEMP_NUM/2;
                NUM_PW=TEMP_NUM-1-NUM_PWB;
                I_UNUSE_BYTE=1024-LENGTH(REAR_REG);
                I_UNUSE_BYTEB=1024-LENGTH(FRONT_REG);
                SUBSTR(P_REG,(PATH_DISP(L)-2-NUM))=
                        LENGTH(PROP_SS)||PROP_SS||
                        SUBSTR(P_REG,PATH_DISP(L));
                P_UNUSE_BYTE=P_UNUSE_BYTE+NUM-LENGTH(PROP_SS);
             END;
          WRITE FILE(TREE) FROM(I_REG) KEYFROM(IREG_NUM);
          WRITE FILE(TREE) FROM(B_REG) KEYFROM(BREG_NUM);
          WRITE FILE(TREE) FROM(P_REG) KEYFROM(PATH_REG(L));
          EQUAL=TRUE;
    END;
END EQUAL_INDX;


COMPUT_SS:PROCEDURE(FRONT_WORD,REAR_WORD);
    IF FRONT_WORD >= REAR_WORD
         THEN DO;
              PUT SKIP LIST('THE ORDER OF WORD IS WRONG');
              STOP;
         END;
    DO J = 1 TO 20;       /* COMPUTE SHORTEST SEPARATOR */
      IF SUBSTR(FRONT_WORD,J,1) NOT = SUBSTR(REAR_WORD,J,1)
             THEN DO;
                  SS = SUBSTR(REAR_WORD,1,J);
                  RETURN;
             END;
    END;
END COMPUT_SS;
```

APPENDIX C

PDL DESCRIPTION FOR MEASURE ROUTINE

```
EXTERN:PROCEDURE;
   DO N= -1 TO (-SIZE_NUM+1) BY -1;
       AREG_NUM=SRREG(N);
       AREG_DISP=SRDISP(N);
       NUM_BLOCK = 0;
       DO WHILE(AREG_DISP < -1);
           NUM_BLOCK=NUM_BLOCK+1;
           READ FILE(ADRS) INTO(ADDR_REG) KEY(AREG_NUM);
           AREG_NUM=RREG(AREG_DISP+1);
           AREG_DISP=RDISP(AREG_DISP+1);
       END;
   END;
   EXT_RATE=TOTAL_EXT*100/(1024*REGION_USED);
END EXTERN;


INTERN:PROCEDURE;
   /* SEARCH THE SMALLEST WORD */
   IREG_NUM=ROOT;
   L = LEVEL;
   DO WHILE(L>1);
      READ FILE(TREE) INTO(I_REG) KEY(IREG_NUM);
      IREG_NUM=SUBSTR(I_REG,5,2);
      L=L-1;
   END;
   LREG_NUM=IREG_NUM;
   /* SEQUENTIAL TRAVERSAL LEAF NODES */
   DO WHILE(LREG_NUM>-1);
      READ FILE(TREE) INTO(L_REG) KEY(LREG_NUM);
      LREG_DISP=7;
      /* COMPUTE UNUSABLE BYTES IN EACH ALLOCATED BLOCK */
      DO N=1 TO NUM_WORD;
         BASE APOINT ON L_REG(LREG_DISP+2+WORD_LEN);
         LREG_DISP=LREG_DISP+4+WORD_LEN;
         CALL DISPLAY;
      END;
      LREG_NUM=HORIZONTAL;
   END;
   RATE=(IN_BYTE*100)/TOTAL_BYTE;
   PRINT 'INTERNAL FRAGMENTATION';
END INTERN;
```

```
DISPLAY:PROCEDURE;
   READ FILE(ADRS) INTO(ADDR_REG) KEY(AREG_NUM);
   BASE ADDR_FIELD ON ADDR_REG(AREG_DISP+1);
   IF NUM_ADDR < 0 THEN RETURN;
   INDX=ISIZE(AREG_DISP);
   NUM_BLK(INDX)=NUM_BLK(INDX)+1;
   IF SIZE(INDX)=1024 THEN TEMP_BYTE=8+4*NUM_ADDR;
                      ELSE TEMP_BYTE=6+4*NUM_ADDR;
   INT_BLK(INDX)=INT_BLK(INDX)+SIZE(INDX)-TEMP_BYTE;
   TOTAL_BYTE=TOTAL_BYTE + SIZE(INDX);
   IN_BYTE=SIZE(INDX)+IN_BYTE-TEMP_BYTE;
   IF NUM_ADDR > 254 THEN DO; /* SPANNED RECORD */
        J=NUM_ADDR/254;
        DO I= 1 TO J;
            ADDR_REGS=ADDR_REG;
            READ FILE(ADRS) INTO(ADDR_REG)
                                    KEY(AREG_NUMS);
            BASE ADDR_FIELD ON ADDR_REG(AREG_DISP+1);
            INDX=ISIZE(AREG_DISP);
            NUM_BLK(INDX)=NUM_BLK(INDX)+1;
            IF SIZE(INDX)=1024 THEN TEMP_BYTE=8+4*NUM_ADDR;
                               ELSE TEMP_BYTE=6+4*NUM_ADDR;
            INT_BLK(INDX)=INT_BLK(INDX)+SIZE(INDX)-TEMP_BYTE;
            TOTAL_BYTE=TOTAL_BYTE + SIZE(INDX);
            IN_BYTE=SIZE(INDX)+IN_BYTE-TEMP_BYTE;
        END;
  END;
END DISPLAY;
```

APPENDIX D

TEST CASE SIZE TABLES

TABLE VII

SIZE TABLES FOR BINARY, FIBONACCI, AND WEIGHTED BUDDY SYSTEMS

BINARY		FIBONACCI		WEIGHTED	
SIZE	K	SIZE	K	SIZE	K
16	0	12	0	8	0
32	1	12	0	12	0
64	1	40	0	16	0
128	1	52	3	24	3
256	1	64	3	32	4
512	1	104	3	48	3
1024	1	156	3	64	4
		220	3	96	3
		324	3	128	4
		480	3	192	3
		700	3	256	4
		1024	3	384	3
				512	4
				768	3
				1024	4

TABLE VIII

SIZE TABLES FOR GENERALIZED FIBONACCI BUDDY SYSTEMS

1		2		3		4	
SIZE	K	SIZE	K	SIZE	K	SIZE	K
16	0	16	0	16	0	16	0
24	0	24	0	28	0	32	1
40	2	40	2	44	2	48	2
64	2	64	2	72	2	80	2
88	3	104	2	100	3	112	3
128	3	168	2	116	5	160	3
256	1	272	2	160	4	208	4
384	2	376	3	276	2	320	3
640	2	480	4	348	5	432	4
1024	2	584	5	464	4	544	5
		752	5	624	4	704	5
		920	6	724	7	864	6
		1024	8	824	8	944	9
				924	9	1024	10
				1024	10		

VITA 2

HWEY-HWA WUNG

Candidate for the Degree of

Master of Science

Thesis: AN EXPERIMENTAL IMPLEMENTATION FOR PREFIX B-TREE AND ASSOCIATED DYNAMIC LISTS

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Taichung, Taiwan, Republic of China, March 22, 1954, the daughter of Mr. and Mrs. C. S. Wung.

Education: Graduated from Taipei Municipal First High Girls' School, Taipei, Taiwan, Republic of China, in June, 1972; received Bachelor of Education degree from National Taiwan Normal University, Taipei, Taiwan, Republic of China, in June, 1978; completed requirements for the Master of Science degree at Oklahoma State University in July, 1983.

Professional Experience: Programmer, Fluid Power Research Center, Stillwater, Oklahoma, Jan, 1981 - Dec, 1981; programmer, Agricultural Economics Dept., Stillwater, Oklahoma, Jan. 1982 - April 1983.